REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (*if applicable*): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*): Monterey, CA 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (*if applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION N

11. TITLE (*Include Security Classification*)
THE INSTRUMENTATION OF THE MULTIMODEL AND MULTILINGUAL USER INTERFACE (U)

12. PERSONAL AUTHOR(S)
Bourgeois, Paul A.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Master's Thesis	FROM 04/90 TO 03/93	March 1993	124

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the offici policy or position of the Department of Defense or the United States Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
Data model, data language, multibackend database system, multilingu database system, multimodel database system, user interface design.

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

The traditional approach to database-management-system (DBMS) designs focuses upon the implementation of single data model and its corresponding data language in order to support applications for a given database task. Eac of these monomodel and monolingual database systems represents a homogeneous database system, since only on data model-language can be supported on a single database system. The application diversity forces many organiz tions to operate several different homogenous database systems to support its operations. A different approach to d tabase-system design is the development of a DBMS which supports multiple data models and their data language This approach is the focus of the multimodel and multilingual database system (MM&MLDS) as implemented on th Multibackend Database Supercomputer (MDBS).

With the proliferation of new data model-languages in the database technology, the objective of MM&MLDS i to incorporate these new data model-languages onto the same MDBS. The goal of this thesis is to develop procedure methods, and tools for the incorporation of new data model-languages into MM&MLDS as new interfaces. Three areas of research are critical to achieving this goal . First, the development of a MDBS user's manual for familia

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: David K. Hsiao	22b. TELEPHONE (*Include Area Code*): (408) 656-2253	22c. OFFICE SYMBOL: CS/Hq

T259815

19. Continued: ization as well as instruction for system users. Second, the specification of generic processing lgorithms used as a foundation for each module of the new model-language interface. Third, our software meth-odology considerations for new data model-language interface implementation.

Approved for public release; distribution is unlimited

THE INSTRUMENTATION OF THE MULTIMODEL AND MULTILINGUAL USER INTERFACE

by
Paul Alan Bourgeois
Captain, United States Marine Corps
B.S. Economics, United States Naval Academy, 1987

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1993

Gary Hughes, Acting Chairman,
Department of Computer Science

ABSTRACT

The traditional approach to database-management-system (DBMS) designs focuses upon the implementation of a single data model and its corresponding data language in order to support applications for a given database task. Each of these monomodel and monolingual database systems represents a homogeneous database system, since only one data model-language can be supported on a single database system. The application diversity forces many organizations to operate several different homogenous database systems to support its operations. A different approach to database-system design is the development of a DBMS which supports multiple data models and their data languages. This approach is the focus of the multimodel and multilingual database system (MM&MLDS) as implemented on the Multibackend Database Supercomputer (MDBS).

With the proliferation of new data model-languages in the database technology, the objective of MM&MLDS is to incorporate these new data model-languages onto the same MDBS. The goal of this thesis is to develop procedures, methods, and tools for the incorporation of new data model-languages into MM&MLDS as new interfaces. Three areas of research are critical to achieving this goal. First, the development of a MDBS user's manual for familiarization as well as instruction for system users. Second, the specification of generic processing algorithms used as a foundation for each module of the new model-language interface. Third, our software methodology considerations for new data model-language interface implementation.

TABLE OF CONTENTS

I. INTRODUCTION 1
A. AN OVERVIEW 1
B. OUR MOTIVATION AND GOALS 3
C. THE ORGANIZATION OF THE THESIS 4
II. THE MULTIBACKEND DATABASE SUPERCOMPUTER (MDBS) 5
A. THE DESIGN 5
B. THE KERNEL DATA MODEL AND ITS KERNEL DATA LANGUAGE 6
C. THE MULTIMODEL AND MULTILINGUAL DATABASE SYSTEM 7
III. PROCEDURES TO INTRODUCE NEW MODEL-LANGUAGE INTERFACES 11
A. JUSTIFICATION 11
B. THE CONTROLLER SOFTWARE 12
C. NEW MODEL-LANGUAGE INTERFACE REQUIREMENTS 14
1. Considerations for the Language Interface Software Module Design 14
a. The Language Interface Layer 14
b. The Kernel Mapping System 16
c. The Kernel Formatting System 18
d. The Kernel Controller 19
2. Communications Among Module 22
3. Data-structure Requirements 25
4. The Makefile Development 27
D. VERSION CONTROL AND MANAGEMENT 27
IV. GENERAL STEPS TO A NEW MODEL-LANGUAGE INTERFACE 29
A. THE MDBS USER INTERFACE 29
1. On the Interface Familiarization 29
2. The User's Manual 30
B. OUR SOFTWARE METHODOLOGY 32
1. Models With A Formal Data Language 33
2. Models With No Formal Data Language 34
3. Kernel Model-Language Mapping Strategies 35
V. THE CONCLUSION 37
A. RESULTS OF OUR RESEARCH EFFORT 37
B. FUTURE RESEARCH 38
APPENDIX A. MDBS USER'S MANUAL 39

APPENDIX B. MODEL-LANGUAGE INTERFACE GENERIC FUNCTION MAPPING 106
APPENDIX C. MM&MLDS GENERIC MODEL-LANGUAGE DATA STRUCTURE 110
APPENDIX D. GENERIC MAKEFILES FOR NEW MODEL-LANGUAGE INTERFACES 111
REFERENCES 113
INITIAL DISTRIBUTION 115

LIST OF FIGURES

Figure 1 : The Multi-Backend Database Supercomputer 6
Figure 2 : The Multimodel and Multilingual Database System 8
Figure 3 : The Model-Language Interfaces as Implemented on a Kernel Database System, MDBS 9
Figure 4: The File Organization and Structure for Data Model-Language Interfaces within the Front-End Controller 13
Figure 5: The Generic Algorithm for All Language Interface Layers 15
Figure 6: The Generic Algorithm for All Kernel Mapping Systems 17
Figure 7: The Generic Algorithm for All Kernel Formatting Systems 18
Figure 8: The Schematic Diagram of Processing Steps of the Kernel Controller 21
Figure 9: The Generic Algorithm for All Kernel Controllers 22
Figure 10: The Module Communication Path for a Data-Definition Load 23
Figure 11: The Module Communication Path for Executing Transactions 24
Figure 12: The User_info Structure 25
Figure 13: The li_info Structure with the new_model_language_info Included 26
Figure 14: The Relationship Between Model/Language Interfaces (ML/Is) and the Test Interface 30
Figure 15: The Lex and Yacc Parsing Processes 34

I. INTRODUCTION

A. AN OVERVIEW

Database systems have revolutionized the workplace by combining the advancements in today's information technology with the information demands of large corporations and governments. These organizations have come to rely heavily upon the efficiency of database systems to increase their productivity. Though the rapid growth that has taken place in the area of database research and development resulting in more efficient database systems and increased productivities, it has also presented a problem due to the proliferation of different database systems, i.e. heterogeneous database systems, that cannot communicate with each other. In examining the factors and issues of the proliferation of heterogeneous databases and systems, we must first review the conventional design of a database system.

The design of a database system begins with the selection of a data model which characterizes the needs of data in order for the processing to be accomplished by the database system. Once an approriate data model (i.e relational, network, or hierarchical) has been selected, a corresponding data language is chosen. Some examples of data models and data languages include SQL for the relational data model, DML for the network data model, and DL/I for the hierarchical data model. Each database system can only support one specific pair of data model and data language . These conventional database systems are characterized as *monomodel* and *monolingual.* A user of such a database system must have thorough understanding of the underlying data model and data language supported by the system in order to use the system effectively. This restricts the user to the *single* data model and data language supported by the system. Any experience with another data model and data language will be useless, since the database system can not support that pair of data model and languag.

A different approach to the database design is the incorporation of several data models and data languages into one database system [DEMU 87]. This more flexible database

design will result in a database system known as the *multimodel and multlingual database system* (MM&MLDS). This approach to the database design provides several distinct advantages not found in the conventional approach to the database design. The consolidation of several data models and data languages on one system will defray the cost spent on procurring several different monomodel and monolingual database systems. The consolidation of resources will also reduce the number of support personnel required to maintain the different database systems. Replacement costs will be reduced since any system upgrade will only be necessary toa single database system instead of several. For example, if one conventional database system is upgraded, the other conventional database systems will not reap the benefits of the upgrade. With the MM&MLDS approach, all data models and data languages will benefit from the upgrade of the single database system.

Re-training costs are also eliminated since a user is not required to learned a new data models and data languages with MM&MLDS. Through cross-modeling access capability, a user who is inexperienced in using a certain new data model-language can access a heterogeneous database with a transaction written in the data model-language more familiar to the user. For instance, a user who is experienced in working with relational databases using the data language SQL could access hierarchical databases using SQL transactions, not DL/I (hierarchical) transactions.

The MM&MLDS concept is currently implemented at the Naval Postgraduate School's Laboratory for Database Systems Research on the Multi-Backend Database Supercomputer (MDBS). MDBS is designed to support MM&MLDS and together, to have the characteristics associated with a federated database system. They include the transparent access to heterogenous databases, local autonomy of each heterogeneous database, and multimodel/multilingual capability. A in-depth look into the database design approach to federated databases and systems can be found in [HSIA 92] and [HSIA 89].

B. OUR MOTIVATION AND GOALS

The MDBS currently implemented at the Naval Postgraduate School's Laboratory for Database Systems Research incorporates four data-model-and-data-language interfaces. These are the attribute-based-data-model (ABDM)-and-attribute-based-data-language (ABDL) interface, the relational-data-model-and-SQL-data-language interface, the network-data-model-and-CODASYL-data-language interface, and the hierarchical-data-model-and-DL/I-data-language interface. Four separate software modules are written for each model/language interface. The process of incorporating a new data model and data language into the current MDBS requires spending numerous man-hours deciphering the data structures and internal logic of previous model-language interfaces.

The first goal of this thesis is, therefore, to provide a pedagogical aid to the incorporation of a new model-language interface into MDBS. As new data models and their new data languages are developed, we want to develop procedures that will ensure accurate and efficient incorporation of their interfaces. We will examine the internal logic common to all interfaces and highlight the necessary functions that a new interface must possess by looking at the four software modules that comprise each interface. Procedures will be presented which outline specific issues that must be addressed prior to any implemention of a new data model-language interface.

The second goal of this thesis is to develop an instructional set or user's manual for setting up MDBS. The manual is designed specifically as an introduction to MDBS for first-time users, but will also function as a reference manual for any further research on MDBS. Since no user's manual currently exists for MDBS, this user's manual will alleviate the need for expert assistance for those who desire to work on MDBS. The MDBS user's manual will address topics and methods such as starting MDBS, developing schemas and request files, and accessing each data model- language interface. Sample databases will be given to assist the user in the understanding of each data model and its data language.

C. THE ORGANIZATION OF THE THESIS

Since all research efforts in this thesis have been accomplished for MDBS, an outline of MDBS is necessary which is presented in Chapter II. More specifically, in Chapter II, we focus upon the system structure, the kernel data model and the kernel data language, and the multimodel-mulitlingual database system design. In Chapter III, we outline the procedures and methods for the introduction a new data model-language interface. Issues such as program and data structure modifications, makefile development, as well as functionality requirements for each module within the model-language interface. In Chapter IV, we address the various interface management strategies when a new data model-language interface is incorporated into MDBS. In Chapter V, we present our concluding thoughts concerning the interface management in the multimodel and multilingual database enviroment, design and implementation decisions for the implemention of a new data model-language interface, and future work.

Appendix A is the first volume of MDBS User's Manual. This manual details those areas concerning the front-end software of MDBS to include the execution of MDBS itself, file development, and the utilization of the data model-language interfaces. Appendix B provides the generic mapping of functions required by all data model-language interfaces as well as specifications for the each software module of the language interface.

II. THE MULTIBACKEND DATABASE SUPERCOMPUTER (MDBS)

A. THE DESIGN

The heart of MDBS is the configuration of the database stores and database processors known as database backends. MDBS relies upon these database backends to provide the storage and processing functions. Each backend consists of a microprocessor and three data disk drives; a smaller Winchester-type for paging and another smaller one for metadata and a larger disk drive for base data. MDBS architecture allows these backends to be connected in parallel via an ethernet LAN using point-to-point communication for one-to-one communications between separate backends and broadcast communication from one backend to many. A front-end microcomputer known as the controller, which is separate from the backends, controls the interface between the user and the backends and also provides for backup and recovery. In Figure 1, we illustrate MDBS architecture [HSIA 91].

Each database backend contains a portion of a stored database through a process called clustering. In clustering, the base data is spread across the backends in mutually exclusive sets. The distribution of the database records through clustering permits parallel access to the data and is integral to the high-performance of MDBS. When a transaction is broadcasted from the controller, each backend can execute the transaction in parallel with the other backends.

The MDBS architecture allows for increases in performance and capacity through the addition of backends. Unlike the traditional database system which required an extensive and costly modification or replacement to achieve a decrease in the response time or a greater capacity without any degradation of the response time, MDBS requires only the addition of one or more backends to achieve such decrease or capacity growth. These performance gains achieved by MDBS through the addition of more backends are known as the *response-time reduction* and the *response-time invariance* [HALL 89]

MDBS provides greater flexibility, since no modification is required to the software which runs on MDBS. The number of backends the system can support is not limited by

connection ports of either the controller or a backend. Therefore, little "down-time" is required for the incorporation of a new backend to the MDBS configuration. At one time, there can be eight backends configured in the Laboratory for Database System Research

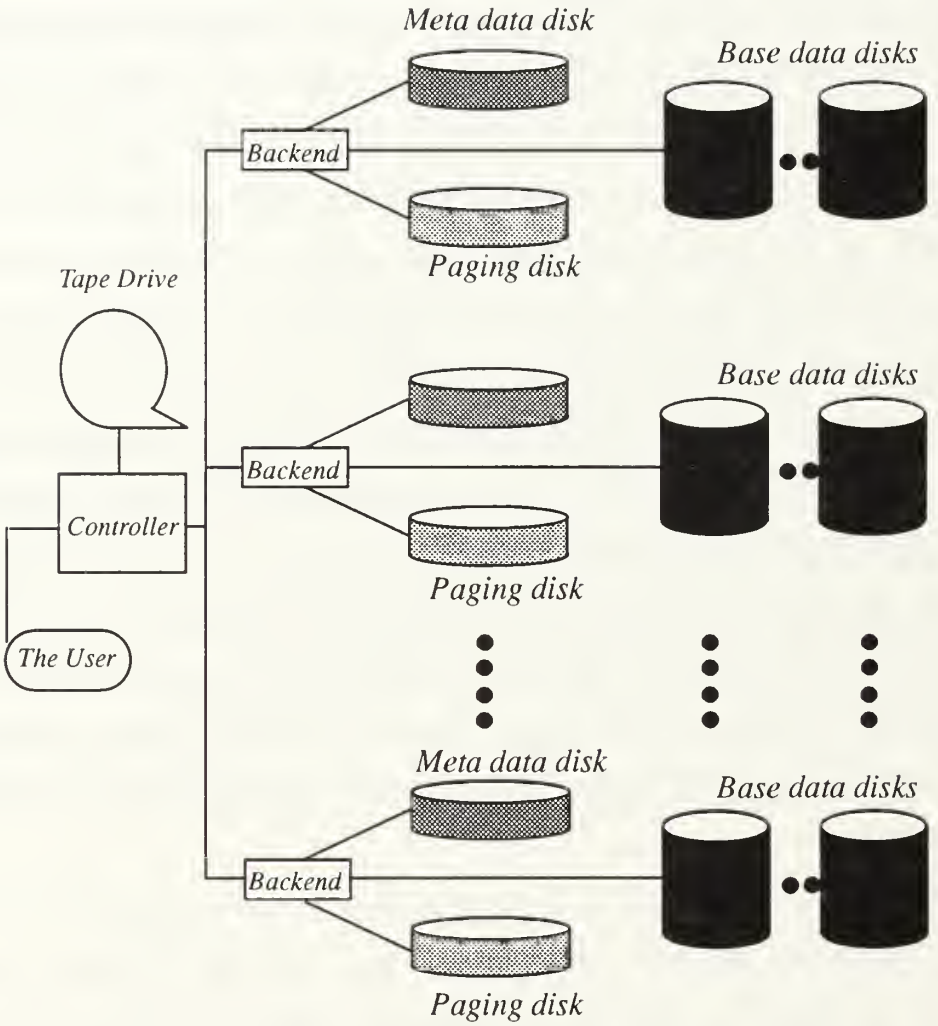


Figure 1 : The Multi-Backend Database Supercomputer

B. THE KERNEL DATA MODEL AND ITS KERNEL DATA LANGUAGE

One of the requirements of a federated database system is to support many data models and their data languages on the same, single system. Such a system is known as being multimodel and multilingual. MDBS is a database system which uses a mapping function

(1) to support its multimodel/multilingual capability and (2) to enhance data sharing amongst different heterogeneous databases. The mapping function used by MDBS maps the different data models and their data languages into a single data model and its data language or kernel data model and its kernel data language. An in-depth discussion of data sharing and mapping algorithms can be found in [HSIA 92].

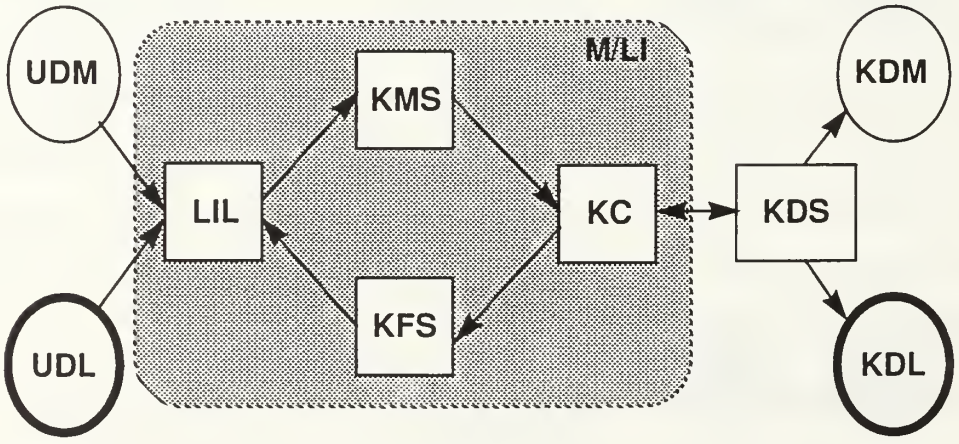
This kernel data model and its kernel data language used by MDBS is the attribute-based data model (ABDM) and attribute-based data language (ABDL). Several factors have led to the selection of the ABDM; these include: the separate modeling of base data and meta data,the clustering of base data into mutually exclusive sets for storage on the backends, and the allowance of parallel accesses to the clustered base data. The semantic richness of ABDL allows for the translation of other traditional data languages, such as SQL, DL/I, and DML into ABDL for processing on MDBS.

Schema definitions and transaction requests developed for traditional data models and their languages are either transformed or translated into equivalent schemas of the kernel data model or equivalent transactions in the kernel data language for processing in the kernel database system. Using the multiple- models/languages to the single-model-language mapping function, MDBS requires only (n-1) schema transformers and (n-1) transaction translators, were n represents the number of different heterogeneous databases and their languages to be supported in MDBS.

C. THE MULTIMODEL AND MULTILINGUAL DATABASE SYSTEM

In Figure 2, we illustrate the general system structure of the multimodel and multilingual database system (MM&MLDS). Transactionswritten in the user's data model (UDM) and user's data language (UDL) are transformed and translated into the kernel data model (KDM) and kernel data language (KDL) equivalent via the MDBS *model/language interface.* Four software modules comprise the model/language interface. These are the language interface layer (LIL), the kernel mapping system (KMS), the kernel formatting system (KFS), and the kernel controller (KC).

The following paragraphs outline the general interaction between UDM and UDL with the language interface as well as KDM and KDL .Each transaction issued in UDM/ UDL must first be processed by LIL. LIL forwards these transactions to KMS. Notice that these transactions may either be database-definition transactions or query-request transactions. Two major tasks are accomplished by KMS. First, when a new database is created, KMS takes the database-definition (or database schema) of UDM and transforms it into a database definition of KDM. This transformation process is called *data-definition transformation*. Upon successful transformation of UDM-database definition, KMS sends KDM-database definition to KC. KC passes KDM-database definition to the kernel database system (KDS) where the new database is finally defined on MDBS. The KC is notified by KDS when the database definition has successfully processed and KC in turn notifies the user through LIL that the database definition has been loaded.



UDM : User Data Model
UDL : User Data Language
LIL : Language Interface Layer
KMS : Kernel Mapping System
KFS : Kernel Formatting System
KC : Kernel Controller
M/LI : Model/Language Interface
KDS : Kernel Database System
KDM : Kernel Data Model
KDL : Kernel Data Language

Figure 2 : The Multimodel and Multilingual Database System

Secondly, KMS translates transactions written in UDL into equivalent KDL transactions through a process known as the *data-language translation*. KMS forwards these transactions to KC which passes them to KDS for execution. Upon completion of the transaction execution, KDS sends the results to KC in KDM format. In order to display the results of a transaction in the proper UDM format, KC passes the results from KDS to KFS. KFS reformats KDM results from KDS into the correct UDM format. Through LIL, the results of the queries are displayed to the user in UDM form.

Each data model-language interface supported by MDBS has its own language interface. In other words, each language interface has its own set of LIL, KMS, KFS, and KC. The functionality, integrity, and consistency of each data model-language must be incorporated within the language interface modules. In, Figure 3, we represent the implementation of the MM&MLDS structure in MDBS.

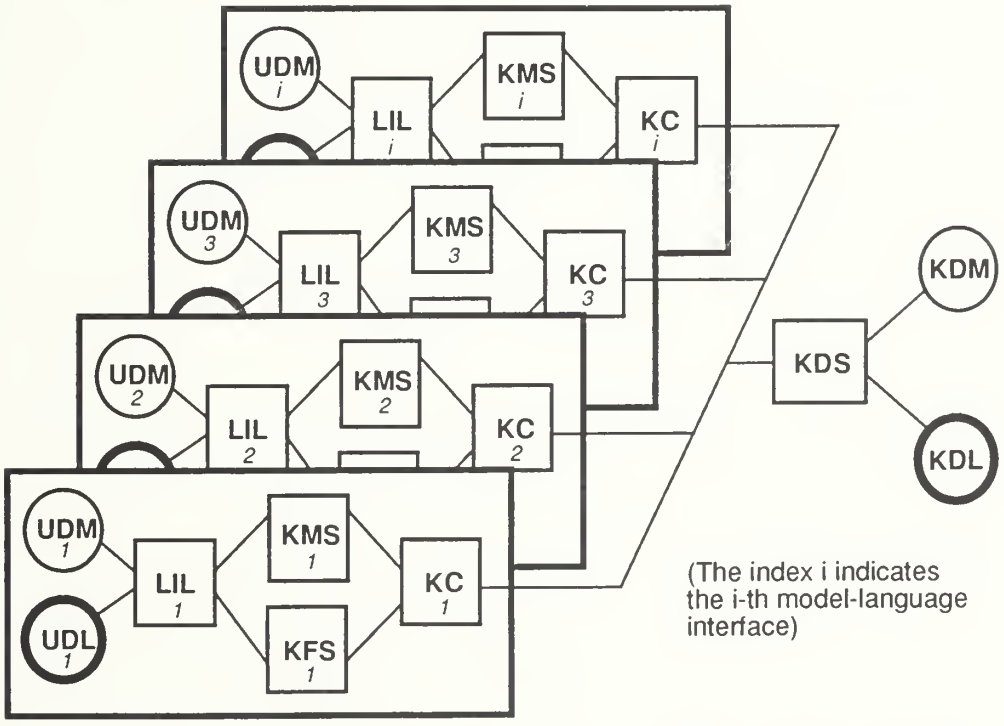


Figure 3 :The Model-Language Interfaces as Implemented on a Kernel Database System, MDBS

Using its capability to model different data models and to translate different transactions languages on one system, MDBS affords the user the opportunity to experiment with different data models and their different data language without prolierating stand-alone database systems. Further, a user may choose the model-language pair which is best suited for the user's needs. Most importantly, the MM&MLDS concept incorporated into MDBS is an invaluable paradigm for the understanding the various data models and their data languages without having to use a separate database system for each data model-language.

III. PROCEDURES TO INTRODUCE NEW MODEL-LANGUAGE INTERFACES

A. JUSTIFICATION

As the database system continues to grow in use and size within government, education, and industry, new data model-languages will be developed to support the constantly changing requirements placed upon existing data model-language interfaces. At the Naval Postgraduate School's Laboratory for Database Research, we would like to be able to incorporate these new data model-languages into MDBS. The introduction of additional data model-language interfaces into the MDBS is not a small task. First, the design and programming teams must first become familiar with the idiosyncrasies of MDBS through hands-on exposure to the current MDBS system. Since a primary focus of designing a new user interface is to maintain consistency throughout each application (or model-language interface in the case of MDBS) [SHNI 92], it is essential that designers and programmers understand the design of previous data model-language interfaces.

Second, once these teams become familiar with the current version of MDBS, an thorough understanding of the general architecture of MDBS is required. This requirement is generally satisfied by reading various papers written on the MDBS. Topics such as federated databases, database architecture, and multimodel/multilingual database design should be covered.

Third, a review of previously designed and implemented data model-language interfaces must be accomplished. Each data model-language interface consists of approximately ten to twenty thousand lines of programming code written in C, and with Lex and Yacc. Few students are familiar with the programming language C, let alone the Unix tools Lex and Yacc. Now, they must learn these languages and tools in order to understand the internal logic of these previously designed interfaces. This review represents a majority of the work that must be accomplished prior to writing the first line of C code for the new data model-language interface.

Previous research has been conducted in the development of traditional data model-languages; however, no research has been done in the area of data model-language interface management on MDBS. Too much time is spent on trying to complete the tasks previously mentioned. The development of an effective tool is therefore necessary, so that the tools will alleviate the burden placed upon designers and programmers and is integral for an efficient and accurate data model-language incorporation. Currently, no such tool exists.

The scope of this data-model-language-interface management strategy is threefold. First, a user manual must be developed that introduces first-time user to MDBS and is also functional as a quick reference for experienced users. Second, for each software module of the language interface contains procedures that are common to all data model-language interfaces, an awareness of these features to designers and programmers is accomplished by the introduction of generic software module algorithms. Third, general steps must be provided for incorporating new data model-language interfaces into MDBS. It is through the fulfillment of this strategy that we are able to provide the paradigm for the instrumentation of the multimodel and multilingual user interface.

B. THE CONTROLLER SOFTWARE

The MDBS configuration separates the controller software from the backend software. All MDBS software is loaded under in a directory named after the most current version of the MDBS software. Currently that version is VerE.6. Subdirectories under the VerE.6 directory contain the software for each particular aspect of MDBS execution. The CNTRL directory contains the software for the data model-language interfaces as well as the software for the communications between the front-end controller and the backends. Our research focus is placed upon the software which comprises the data model-language interfaces. This software is located in the TI (for Test Interface) directory and subsequent subdirectories of TI. Figure 4 illustrates the file organization of the test interface directory and its relationship with the data model-language interface.

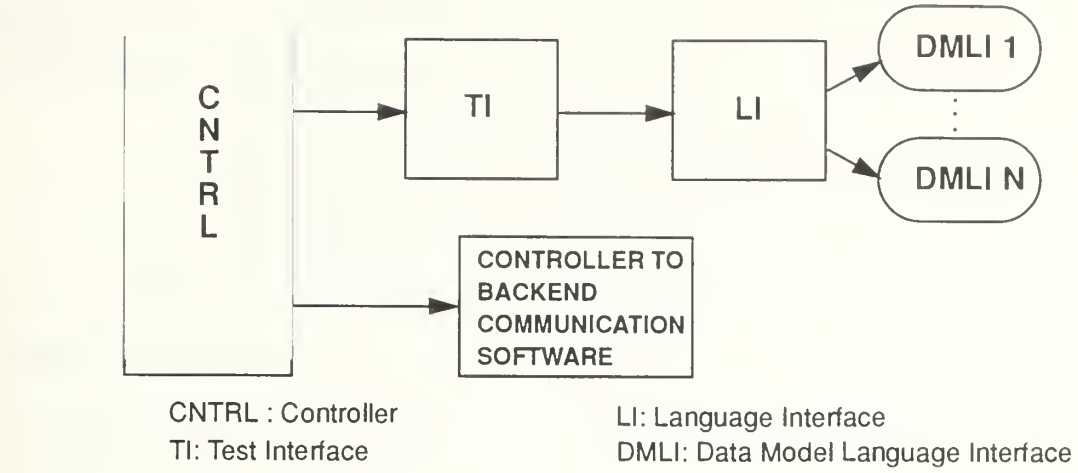


Figure 4: The File Organization and Structure for Data Model-Language Interfaces within the Front-End Controller

Within the TI directory resides the program module which drives the user interface for MDBS. This program, ti.c, determines the number of backends the system has currently configured for execution, loads the schemas for existing database schemas, and displays the MDBS system menu. Implementors of new data model-language interfaces must include an option in the *main (argc, argv)* procedure of *ti.c* for the selection of the new data model-language interface. This includes adding the appropriate case statement to allow continued processing of the new model-language interface. Appendix B provides a detailed outline of the test interface directory structure as it pertains to the MDBS user interface.

The TI directory also contains the procedures for executing the kernel data model-language interface, i.e., attribute-based data model-language interface. The introduction of a new model-language interface does not require any modification of the programs residing in the test interface directory with the exception of ti.c as mentioned above.

In order to alleviate the burden of data passing among functions residing in different software modules, MDBS relies upon global data structures to communicate between software modules within the language interface. Within the TI directory, global data-structures and variables are stored in a header file named licommdata.h. Part 4 of Section

C outlines the data structures required by all model-language interfaces and their relationship to the licommdata.h file. Appendix C diagrams the generic data-structure relationship required by all new model-language interfaces.

C. NEW MODEL-LANGUAGE INTERFACE REQUIREMENTS

1. Considerations for the Language Interface Software Module Design

As previously noted, the language interface consists of four software modules: the language interface layer, kernel mapping system, kernel formatting system, and kernel controller. Each new data model-language introduced into MDBS must include its own set language interface modules; specifically, its own LIL, KMS, KFS, and KC. Even though each data model-language interface implemented on MDBS has its own unique set of language interface software modules, certain functions must exist in all model-language interfaces. It is these required functions and corresponding internal logic that must be addressed when designing and implementing a new data model-language interface on MDBS. This is accomplished by examining the language interfaces of existing data model-languages and identifying the commonality that is present in each module.

a. The Language Interface Layer

The language interface layer (LIL) of the MDBS functions as the bridge between the user and the controlling software which drives MDBS. Regardless of the model-language interface implemented on MDBS, all model-languages interfaces communicate with the user via LIL. Communications with the kernel mapping system, kernel controller, and kernel formatting system are also the responsibility of LIL.

Since all model-language interfaces interact with the user in LIL, there must be some form of consistency among the different user interfaces. The goal of the user interface design for MDBS is to develop a user interface that is common for each different data model-language interface. Achieving this goal reduces the time required by the user to familiarize the user with the functionality of a different user interface with each data model-language present on MDBS. For instance, a user familiar with the relational/SQL user

interface does not have to learn a completely different user interface in order to use the hierarchical/Dl/1 interface.

Each data model-language possesses a similar pattern of processing for the LIL We require that LIL is to adhere to a generic algorithm for the interaction with the user as well as KMS, KC, and KFS. New model-language interfaces must incorporate this generic algorithm into its own language interface layer. Figure 5 outlines the structure of the generic LIL processing algorithm.

- Allocate and initialize memory for data-structures.
- Process new or old database?
- If new database...
    - Enter and store database name into the database catalog.
    - Load schema input from the terminal or file.
    - Load the file into linked-list.
    - Create template and descriptor files, index attributes
    - Send schema (in linked-list data structure) to KMS for parsing.
        - Display errors from KMS during parsing (if any).
    - Prompt to continue to process as old database or exit.
- If old database...
    - Check database name against the catalog. If not found check other model-language catalogs (this is for use with cross-model accessing only).
        - if found in other data model-language, transform schema.
    - Determine the mode of the input request.
    - Store requests in a linked-list data structure.
    - Display numbered queries to the user.
    - Option to redisplay queries or process a request.
    - Send the list of requests to KMS for parsing.
    - Display errors from KMS (if any).
    - Send the parsed request to KC for processing against a database in KDS.
    - Display results from KFS to user.
    - De-allocate the dynamic memory.
- Exit to main menu and repeat above steps or exit system

Figure 5: The Generic Algorithm for All Language Interface Layers

Notice that this algorithm applies directly to any data model-language interface implemented on MDBS. The language interface layer is by far the easiest layer to develop from a programmers stand point. Much of the code already developed from previous model-language interfaces can be modified to suit the requirements of the new model-language interface. The application of the generic algorithm guarantees consistency amongst the various model-language interfaces as well as the basic framework by which the LIL functions.

b. The Kernel Mapping System

The kernel mapping system is responsible for the parsing of the model-language based transactions for further processing in the kernel controller. This parsing involving the transformation and translation of transactions written in the user's data model-language into the data model-language of the kernel database system. Though transformation and translation of UDM-L takes place in KMS, the user is unaware that this transformation and translation is being conducted. This is known as the transparency and is one of the requirements of a federated database system. Designers and programmers of new model-language interfaces must ensure that KMS does not interface directly with the user but instead interfaces with the user through LIL. Again, this is to maintain consistency and transparency throughout the model-language interfaces.

A majority of KMS is comprised of mapping and formatting algorithms that are specific to the model-language implemented. KMS must read input streams from LIL containing either data-definition or request transactions. In order for the input streams to be readable by KDS, KMS uses the grammar and semantics of UDM and UDL to transform or translate the input stream into tokens recognizable by KDS. Functions written in KMS must accommodate the semantics, syntax, and grammar of the model-language being introduced. By analyzing the semantics, syntax, and grammar of UDM and UDL for translation and transformation into KDM and KDL, the parser is not only capable translation and transformation, but is also capable of detecting errors in the transaction

Even though the grammar and semantics of UDM and UDL dictate the transformation and translation process, certain functions are required by KMS in all model-language interfaces. A generic algorithm which outlines the basic functions required of KMS for all model-languages is proposed. Implementation and strict adherence to this algorithm will alleviate the burden of determining what is required by KMS in relation to the other modules of the new language interface.

Figure 5 outlines the generic KMS processing algorithm required in all model-language interfaces on MDBS. Notice that this algorithm is independent of the model-language interface on which it is implemented.

```
- KMS called by LIL to process transactions
- Begin parse of the input stream.
- If error found in parsing
    - deallocate the memory allocated for the data-definition or request
      input streams. This memory is generally the memory allocated by
      the model-language kms_info data-structure.
    - issue the error message.
- If no errors found
    -if a data-definition transaction to load a database schema
        - send transformed schema to KC to load onto KDS.
        - send LIL the confirmation of schema load.
    - if a request transaction for querying.
        - send the translated UDM request to KC.
        - display the translated UDM request in KDM format via LIL
- End processing
```

Figure 6: The Generic Algorithm for All Kernel Mapping Systems

Because KMS represent over half the programming requirement for the language interface, we suggest for a new model-language interface, three steps in the design of the kernel mapping system. We also focus on the mapping and formatting process. These steps function as a guideline for the design of KMS prior to programming.

1. Outline the grammar and semantics of the new model-language interface.
2. Determine the mapping algorithm to mapping grammar and semantics of UDM and UDL into KDM and KDL.

3. Incorporate the mapping and formatting functions into the generic KMS algorithm.

c. The Kernel Formatting System

The primary function of the kernel formatting system (KFS) is to reformat KDM-formatted transactions received from the kernel controller (KC) into the proper UDM format. Communications with KFS are only established after the successful completion of a RETRIEVE or RETRIEVE-COMMON transaction in KC. Other transactions issued to KC only require notification to the user of successful or unsuccessful execution. However, the RETRIEVE and RETRIEVE-COMMON transaction display data results to the user and therefore must be transformed into a format familiar to the user (i.e., in UDM).

Each model language has a unique representation of an ABDL request in regards to the syntax and structure of the data model. Therefore, KFS simply parses the result data stream (or response) from KC in attribute-based form and displays it to the user in the UDM form.

Of the four modules, KFS provides only one simple task to the MDBS transaction execution process and thus comprises the least coding required to implement. Though simple in coding, specific steps must be found in every KFS. This does not prevent the new model-language designer from creating a complex display screen for data having been output by MDBS. With this in mind, we propose a generic kernel controller algorithm for the implementation in all KFS modules. Figure 7 illustrates the generic KC algorithm.

- Receive an output stream from KC.
- Correlate ABDL request data with UDM attributes.
- Display results to the user.
- Return to LIL

Figure 7: The Generic Algorithm for All Kernel Formatting Systems

d. The Kernel Controller

The intersection of the language interface and the back-ends of MDBS meet at the kernel controller (KC). Without a thorough understanding of KC 's internal logic coupled with the utilization of a generic algorithm outlining the functional requirements of KC, the implementation of a new model-language interface would be virtually impossible.

Once a user's data definition or request transaction has been transformed or translated by the kernel mapping system (KMS), the control is passed to the kernel controller for the loading of this (or these) transaction(s) onto MDBS. KC relies upon KMS to issue it the transactions that meet the following criteria:

1. of semantic correctness,
2. of syntactic correctness,
3. in proper KDM format.

The operations performed by KC are based upon the operation flag within the new_model_language_info data structure. Two situations present themselves to the kernel controller. First, the introduction of a new database into MDBS sets the operation flag to CreateDB. This sends a signal to the kernel controller indicating that a data definition has not been loaded for this particular database. Since KMS has transformed UDM data definition into KDM form and LIL has created template and descriptor files for the new database, KC must now load the template file onto MDBS backends. The database template file is required by KDS in order to conduct the initial database load. In the attribute-based data model-language, the template file represents the data definition used by the supported data-model language to establish a new database on the system.

As previously mentioned, functions which comprise the attribute-based data model-language reside in the Test Interface section of the controller software. The Test Interface provides the communication link from the language interface to the backends via the controller. Other modules with the controller aid in the communication of the backends to the controller and vice versa.

KC must call the *dbl_template()* function in the program *dbl.c* in the Test Interface. This function copies the template file into the *UserFiles* directory and then starts the process of loading the template file onto the MDBS back-ends. Any errors encountered during the load of the template file will be displayed to the user and all corresponding dynamic memory will be de-allocated.

The second situation involves translated transactions from KMS awaiting processing by KC. Each request transaction has an associated operation flag assigned upon completion of KMS parsing function. It is this flag which determines what processing functions must be performed by the kernel controller. Operation flags are model-language specific, but must incorporate the five basic operations of the attribute-based data language (ABDL): RETRIEVE, RETRIEVE-COMMON, DELETE, INSERT, and UPDATE. Depending on the operation specified in the operation flag, the appropriate request_handler within KC will be called. If the request transaction required an ABDL translation of more than one ABDL statement, a *request_handler* function must be invoked to ensure continuity between the multiple ABDL requests and the single UDM requests from which they were parsed. This will prevent an incomplete execution or the original UDM request. Aside from the this scenario, the *request_handler* will pass control to the *new_model_langauge_execute* function.

The *new_model_langauge_execute* function within KC makes the necessary function calls with the Test Interface section for subsequent execution on the MDBS processors. The translated UDM request is first modified into the proper ABDL format for execution by changing all characters in the request to lower case with the exception of the first character of the request. Upon completion, the modified request is loaded to the MDBS for execution by calling the *TI_$$TrafUnit()* function in the Test Interface.

After loading the request, KC checks for responses from KDS which may contain the results from previously loaded queries. This is accomplished by invoking the *nml_chk_response_left()* function within KC. Maintaining a count of the number of ABDL

requests needed for a single UDM request, the *nml_chk_response_left()* function receives messages (or responses) from KDS and ensures all ABDL results have been received from KDS prior to sending the results to the kernel formatting system (KFS). Pending the completion of all the requests or transactions, a file is maintained that appends the results of each query as it finishes its execution. Errors in processing will be addressed to the user through the *TI_ErrRes_Output()* function. Successfully processed requests will be forwarded to KFS for display to the user. Figure 8 shows a schematic diagram of the kernel controller processing steps.

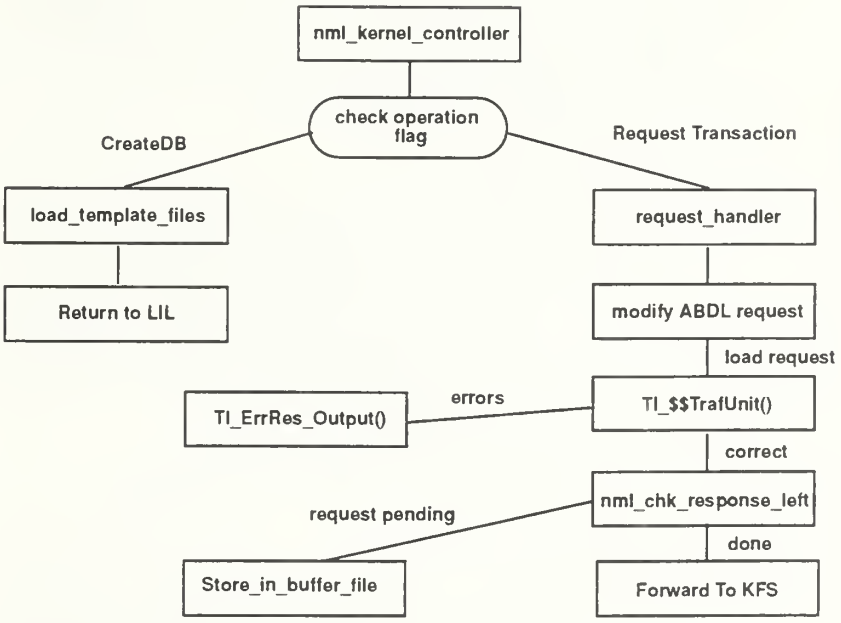


Figure 8: The Schematic Diagram of Processing Steps of the Kernel Controller

Data model-language specifics do not have a major impact upon the design of the KC because model-language transactions reach KC have already been translated into KDM form. Therefore, all KCs must have a similar algorithm which processes the translated UDM transaction and extracts the results from KDS. We propose a generic KC

algorithm required in all model-language interfaces. This algorithm provides the basic foundation by which all KCs should be developed. Figure 9 illustrates this algorithm.

```
- Receive the parsed transaction from KMS via LIL.
- Determine the subsequent action based on the operation flag.
- if operation_flag = CreateDB
    - dbl_template() /* in Test Interface to load template file.
    - return to LIL.
- if operation_flag = a request transaction
    - request_handler() /* based on the operation_flag set in KMS.
    - nml_execute()
        - fix_ABDL_req() /* modify the ABDL request to proper format.
        - TI_$$TrafUnit() /* loads the ABDL request to MDBS.
        - nml_chk_response_left() /* have all requests processed?
        - TI_R$Message() /* receive message from MDBS.
        - TI_R$Type() /* is the message correct?
        - switch (TI_R$Type)
            case correct response:
                - TI_R$ReqRes() /* receive the results.
                - check_last_response() /* are there results pending?
                case RETRIEVE or RETRIEVE COMMON:
                    - if results pending, store results in the buffer and
                      loop till all request results are completed.
                    - KFS() /* forward results to KFS.
                case INSERT or UPDATE or DELETE:
                    - print the query-completion notification.
                    - return to LIL
            case Errors
                - TI_R$ErrorMessage() /* finds the error type.
                - TI_ErrRes_Output() /* display the error to user.
                - return to LIL.
```

Figure 9: The Generic Algorithm for All Kernel Controllers

2. Communications Among Modules

Figure 2 illustrated the basic relationship between the software modules comprising the language interface. For the design and implementation of new model-language interfaces, this figure must be expanded to provide a more in-depth view of the

communication paths that exist between modules. Additionally, the role of the Test Interface must be included to show the relationship the Test Interface has with the Language Interface for all data model-languages.

The path selection is based upon whether a user is loading a data-definition for a new database or loading request transactions for processing against an existing database. If a data-definition is to be loaded, the following intercommunication takes place between the modules and the Test Interface. First, the user issues a command to load a new data-definition via LIL; LIL then queries the user for the database name and uses the name to request for the data-definition file. After receiving the data-definition file, LIL calls KMS to parse the data-definition file into the KDM format (see step 1. in Figure 10). Once completed, the control is returned to LIL (step 2.) which then makes a call to KC (step 3.)for loading of the KDM data-definition to KDS. KC calls the Test Interface to perform the actual loading of the data to KDS (steps 4. & 5.). After KDS has successfully loaded the data-definition to MDBS and notified the Test Interface (steps 6. & 7.), KC returns the control to LIL (step 8.). This completes the loading of the data-definition file. Figure10 illustrates the aforementioned communication sequence with sequence numbers.

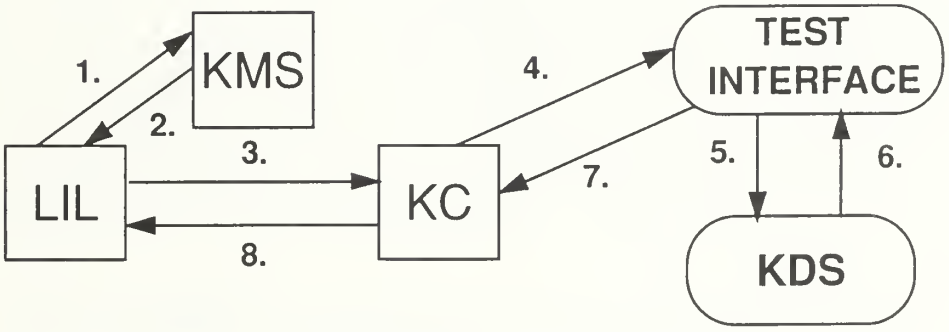


Figure 10: The Module Communication Path for a Data-Definition Load.

Notice that KFS is not involved when loading a data-definition file. KFS is not required in the communication process, since there are no (request) transactions to be

displayed to the user. The omission of KFS is the distinguishing feature of this communication path.

Our second communication path involves the issuing of (request) transactions for the execution in KDS. After the user has responded to the prompt by loading (request/ query) transactions via LIL, those transactions are sent to the KMS for parsing into the KDM form (step 1. in Figure 11). KMS passes the parsed transaction(s) to KC (step 2.) which in turn sends the transactions to the Test Interface (step 3.) for execution on KDS (step 4.). After processing against the appropriate database on MDBS, results are forwarded to the Test Interface (step 5.) in order to further communicate the results to KC (step 6.). KC forwards the results to KFS for display to the user in the UDM form (step 7.). Controls are then finally returned to LIL (step 8.) where the process is repeated if more requests are processed. Figure 11 illustrates the communication path for the execution of (request) transactions in MDBS.

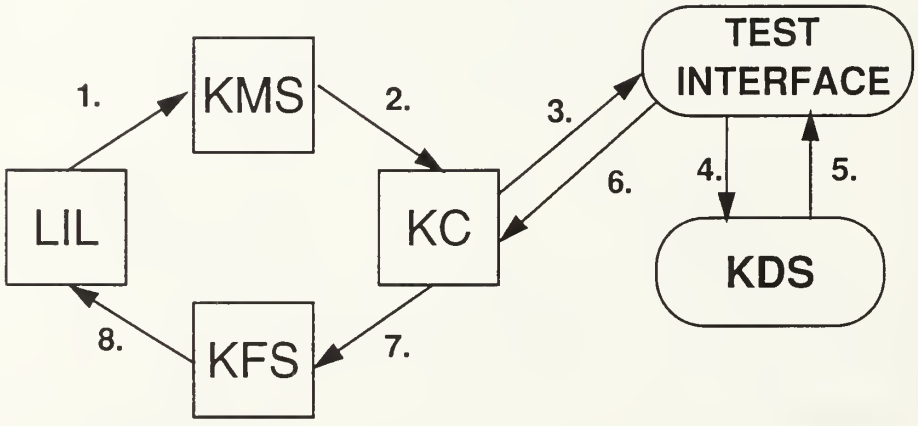


Figure 11: The Module Communication Path for Executing Transactions.

In both cases, errors resulted in any module will cause the control to be returned to LIL. Error messages will be displayed to the user by the module where the error was detected.

3. Data-structure Requirements

The data-structure design for the representation of a new data model-language must incorporate features of both the model-language and MDBS. To accomplish this, we propose a generic data-structure which satisfies the requirements of the language interface and allows expansion for features of the new model-language interface. In order to understand where the generic data-structure fits into MDBS, one must become familiar with data-structure representation in the programming language C.

Any data-structure development on MDBS requires an understanding of derived types in the programming language C. Two derived types, *structure* and *union*, are essential in the data-structure design for MDBS. Structures aggregate data of different types (derived types included) into a single data type. Unions allow alternative values to be stored in a single shared portion of the memory. Integrated into complex data-structures, these derived types are useful in representing the new data model-language. A detailed explanation of these derived types, with examples, can be found in [KELL]

The structure *user_info*, shown in Figure 12, is the building block of the data-structure network which comprises each model-language interface. This structure contains a variable of the type union, *li_info*, which stores the data structure of the model-language interface selected for execution. The implementation of new model-language interfaces into MDBS requires the modification of *li_info* to include a new structure for the new model-language interface as shown in Figure 13. This new structure is our proposed generic data-structure for new model-language interfaces.

```
struct                                  user_info
    {
        char      ui_uid[UIDLength +1];
        union     li_info       ui_li_type;
        struct    user_info     *ui_next_user;
    }
```

Figure 12: The User_info Structure

```
union                                                    li_intc
    {
        struct    sql_info    li_sql;
        struct    dli_info    li_dli;
        struct    dml_info    li_dml;
        struct    dap_info    li_dap;
        struct    new_model_language_info    li_nml;
```

Figure 13: The li_info Structure with the new_model_language_info Included

The generic data-structure for new model-language interfaces is a C structure called *new_model_langauge_info*. This structure contains all the pertinent information required by MDBS, including:

1. Unions for the new model-language KMS, KC, KFS, and LIL.
2. Operation flags to indicate user requests, operations, and errors.
3. Structures for storing filenames, transactions, and data-definitions.

Specifics of the new model-language should also be included in the *new_model_langauge_info* structure. The unions for KMS, KC, and KFS as well as the structure types specific to the features of the selected model-language are stored therein. Appendix C illustrates the generic data structure *new_model_langauge_info* and its relationship to other data structures within MDBS.

All language-interface data-structures reside in the header file licommdata.h in the Test Interface portion of MDBS. New model-languages must store their data-structures in this file and use the C construct *include* in all programs of the new model-language interface. Using a common file for the declaration of model-language data-structures is especially useful when designing a cross-model access capability into a model-language interface since each model-language has the access to the data-structures of the other model-languages.

4. The Makefile Development

As with all programs written in the programming language C, a *makefile* is used to compile several related programs in one step.The programs which comprise a new model-language interface are no exception. Each module in the language interface of a new model-language should contain its own makefile for separate modular compilation. In other words, LIL, KMS, KFS, and KC should each have a separate makefile.

After the development of LIL, KMS, KC, and KFS, a single makefile should be developed for the new model-language interface in order to compile all the modules at one time. The makefile for the new model-language should include a makefile for each module of the new model-language and indicate the appropriate directory to store all object code resulting from the compilation process. After compiling the new model-language interface, the Test Interface must also be recompiled due to changes made to the program ti.c. These changes are a result of adding a new selection option for the new model-language interface to the MDBS main menu. Appendix D illustrates a generic makefile for a new model-language interface.

D. THE VERSION CONTROL AND CONFIGURATION MANAGEMENT

With the proliferation of new model-languages, the need of the version control will present an area of concern to MDBS system managers. As new model-languages are introduced to the MM&MLDS, new versions of the MM&MLDS software must be created and managed. A strategy must be developed to ensure the proper documentation and maintenance of its software when new versions of the MM&MLDS software is developed. Coupled with the version control is the configuration management which must also be addressed.

The implementation of a new model-language interface results in the creation of a new version of the MM&MLDS software. The latest version of the MM&MLDS software should be the only version residing on the system after testing and evaluation is completed.

Utilizing the tape drive on the front-end controller, back-up copies of the old version and new version should be made in case of a catastrophic system failure.

The version control is not limited to changes or additions to the MM&MLDS software. System upgrades may require the modification of the existing backend software. Therefore, the following steps must be followed when a modification occurs to MDBS software.

1. Back-up old and new versions to a tape. Ensure only new version is resident on the system.
2. Properly label tapes and maintain a logbook indicating what modifications were made and why.
3. Ensure all the MDBS systems receive the updated version of the new software.

Through the version control and configuration management, systems requiring only specific data model-language interfaces can modify the new version of the MM&MLDS software so that only those interfaces needed will be available for the selection. This can be accomplished by deleting the statements and respective case statements for the unwanted model-language interface in program *ti.c* of Test Interface. The system manager can then recompile Test Interface and delete any references to the unwanted model-languages from the makefile. This simple process allows system managers to offer only those model-languages required at his specific site.

Any mismanagement of the MM&MLDS software can be costly. Slight modifications (especially to Test Interface) can result in minor errors to a system failure. Following the aforementioned version control and configuration management strategy, we will minimize these problems.

IV. GENERAL STEPS TO A NEW MODEL-LANGUAGE INTERFACE

A. THE MDBS USER INTERFACE

1. On the Interface Familiarization

The current version of the MM&MLDS software utilizes C shells and scrolling menus for user interactions with MDBS. Prior to the design and development of a new model-language interface, a period of familiarization is needed to understand the idiosyncrasies of the MDBS user interface. Without such experience, future model-language interfaces will be constructed in their own unique fashion. An absence of standardization among interfaces will exist on MDBS. This will force users to have to understand and learn the processing steps of several different user interfaces, in addition to the semantics, syntax, and grammar of the new data model-languages. It is because of this problem, standardization of the user interface is a must.

After establishing a standard, the problem we encounter is adhering to that standard and maintaining a level of consistency among the different model-language interfaces. Our familiarization with previous model-language interfaces will provide the designers of the new model-language interface with a guide for the user interface development. Menus appearing to the user in one data model-languages must appear to the user in the other model-language interfaces. Of course, some variation must be allowed for the specifics of the model-language. For example, references to sets is appropriate for an network/DML interface; however, this is not the case for an object-oriented interface. Our focus is placed on the general framework of the model-language interface with the specifics of the model-language incorporated into this framework. Maintaining a level of consistency amongst the different model-language interfaces will reduce both errors and the confusion on the part of the user when alternating between different model-language interfaces. Unique model-language interfaces will detract from the concepts of multimodel and multilingual interfaces instead of enhancing the users understanding of the overall

concept of MDBS. In Appendix D, we illustrate a basic framework for model-language interface design.

When we discuss standardization of model-language interfaces, we exclude the user interface associated with the attribute-based data model-language. The attribute-based data model-language interface software resides outside the model/language interface portion of MM&MLDS. Since all model-language interfaces are either transformed or translated into the attribute-based data model-language, those functions specific to this kernel data model-language should be separate from those model-language interfaces. The attribute-based data model-language interface contains the functions that allow direct communication with the backends unlike other model-language interfaces that communicate with the backends via the attribute-based model-language interface. In Figure 14, we show the interface relationship between the model-language interfaces and the attribute-based model-language interface in regards to backend communication.

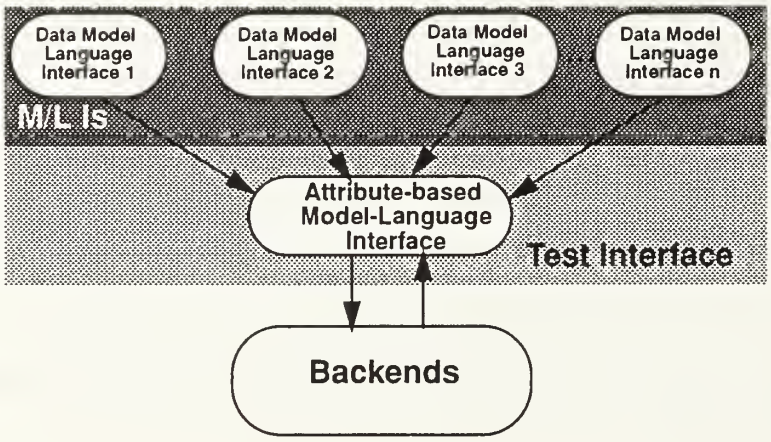


Figure 14: The Relationship Between Model/Language Interfaces (ML/Is) and the Test Interface

2. The User's Manual

MDBS addressed the concept of federated databases and solutions to the problems presented by heterogeneous databases. Therefore, in a classroom or research

environment, MDBS offers an excellent opportunity to (1) familiarize students with the various data model-languages supported on MDBS and (2) allow researchers a vehicle by which to further study the federated and heterogenous database concepts. Unfortunately, in the past, potential users required the expert assistance in order to properly operate MDBS. This placed a tremendous burden upon the user. The user had no written documentation of how to operate the system and therefore was forced to learn the system by trial and error. This greatly diminished the learning objectives of MDBS, since the user was more focused on figuring out how the system operated through trial and error. This lead to frustration and confusion on the part of the user.

In order to alleviate these problems, we propose an instructional set or user's manual which outlines every possible aspect of the user interaction with MDBS. Our intentions were to design a manual that would be beneficial to both first-time users as well as experienced users. Since MDBS is such a valuable resource for instructional purposes, the user's manual is also designed as a teaching aide for advanced database topics. Therefore, a practical vice theoretical lab would be possible, giving the students a more dynamic indoctrination into advanced database topics in addition to lectures. At this time, students in the advanced database course CS4312 are using the MDBS User's Manual to further their knowledge of database concepts.

The introduction to the MDBS User's Manual, found in Appendix A, is the first step in the design and implementation of a new model-language interface on MDBS. This manual is an excellent tool for familiarizing oneself with the idiosyncrasies of MDBS that is required in the design of new model-languages. Topics covered in the MDBS User's Manual are:

1. Multimodel and multilingual capabilities.
2. System overview.
3. Starting the system
4. Data-definition and request file development.
5. Kernel data model-language interface

6. Execution of SQL, DML, and DL/I interfaces.
7. Execution of the cross-modeling access capabilities.
8. UNIX aliases and C shells.
9. Sample databases for the user familiarization.

As new model-languages and new functions to existing model-languages are introduced to MDBS, additions and deletions to the user's manual must be made. This in effect makes the MDBS User's Manual a living document that can be updated as the system grows. Furthermore, as students use the manual as a part of their classroom instruction, clarifications and modifications can be made to further enhance the learning process. Through proper maintenance, the MDBS User's Manual is the most effective paradigm for instructing MDBS users.

B. OUR SOFTWARE METHODOLOGY

The design and implementation of a new model-language interface involves several software design decisions that must be made to ensure effective model-language interface introduction into MDBS. These decisions are a result of developing the best possible strategy needed for incorporating the idiosyncrasies of MM&MLDS with the grammar, syntax, and semantics of the new model-language interface. Of the four modules comprising the Model/Language Interface, the development of KMS involves the greatest amount of planning and preparation since it is in this modules where the grammar, syntax, and semantics of the new model-language interface must be addressed. The other three modules rely upon the design of KMS as a building block for their own design. With this in mind, we introduce our software methodology for the design of a new model-language interface. In order to understand our methodology, three areas critical to the design of the model-language interface software must be addressed. These are (1) designing a new interface for data models with no formal data language, (2) designing a new interface for data models with a formal data language, (3) mapping strategies using the kernel model-language.

1. Models With A Formal Data Language

The introduction of standardized data model-languages into MM&MLDS requires the development of an implementation strategy that encompasses all specified constraints of the model-language. Since there is the documentation which specifies the constraints of the model-language, the designers and programmers are not required to develop the constraints for new model-language. Instead, they have the opportunity to avail themselves a useful programming tools to parse the data model and its data language to ensure complete coverage of all possible constraint violations.

These programming tools, Lex and Yacc, allow parsing of an input stream in the new data model-language for future conversion into the kernel data model-language. Lex and Yacc are useful, since their modules can be incorporated directly into the programming language C and thus be included in the KMS portion of the Model/Language Interface. Lex is a lexical analyzer which parses the input stream into recognizable tokens, these tokens are then input into a compiler create by Yacc. Yacc, which stands for Yet Another Compiler-Compiler, creates a compiler which accepts the tokens from Lex and processes them in a finite-state automaton. Since the data model-language constructs are standardized, all possible accepting states of a valid string are identified and thus can be incorporated into the finite-state automaton generated by Yacc. Further explanation of the implementation and design of Lex and Yacc can be found in [LESK] and [YACC].

By utilizing Lex and Yacc, we may have a more complete description of the data model-language that is to be introduced into MM&MLDS. If a model-language interface becomes standardized since being implemented on MM&MLDS, the parser within KMS should be converted from its existing parser written in C with a Lex and Yacc parser. This would allow (1) the addition and modification of constraints not found in the original parser and (2) produce a parser that covers the constraints of the standardized model-language. In Figure 15 we illustrate the role of Lex and Yacc in the parsing process of KMS.

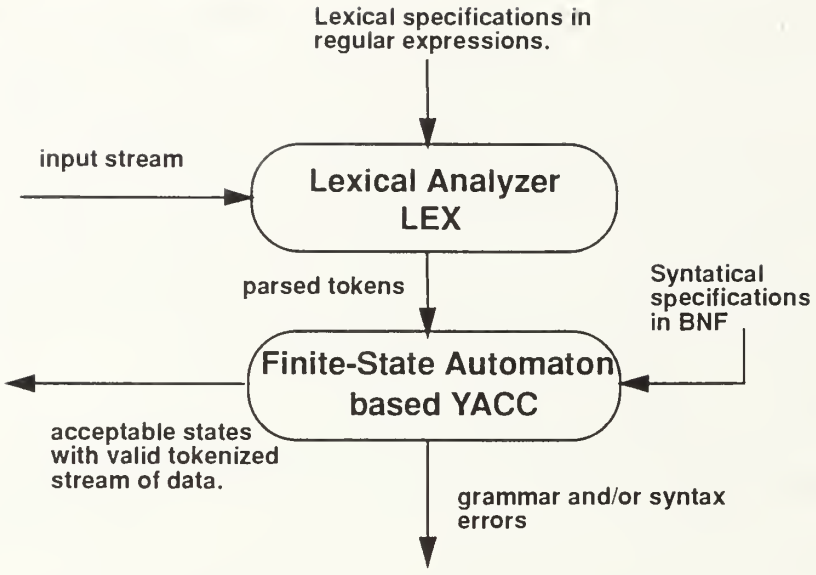

Lexical specifications in regular expressions.
input stream
Lexical Analyzer LEX
parsed tokens
Syntatical specifications in BNF
Finite-State Automaton based YACC
acceptable states with valid tokenized stream of data.
grammar and/or syntax errors

Figure 15: The Lex and Yacc Parsing Processes.

2. Models With No Formal Data Language

With the proliferation of new data model-languages, the standardization of these model-languages may become an involved process especially when no formally accepted standard for the model-language has become available. Such is the case of the Object-Oriented Data Model and Data Language. The object-oriented data model-language concept has been addressed throughout the database community; however, no acceptable standard has been declared. By standard, we mean a data model and its data language that is universally accepted and implemented; e.g. relational and SQL are a standard data model and a data language.

At the Laboratory for Database System Research, we intend to introduce new model-languages into our system whether a data model and its data language have become standardized or not. Without a standardized data model-language to incorporate into the MDBS system, the designers and programmers must develop their own representation of a proposed data model-language. It is this representation that must be developed prior to any

programming efforts to ensure compatibility between the model and its language. A standardized data model and its data language generally evolve from extensive research and testing to ensure all possible integrity constraints are addressed. In the case of data models and their data languages that are not standardized, extensive testing has not yet been conducted and thus the testing requirement falls upon the designers intending to implement the new model-language interface on MDBS. The documentation of integrity violations in standardized data model-languages is a tremendous aid in developing an accurate and efficient parser to parse the data model and its language into the kernel data model-language as addressed in the next section. This aid is not available to the designers of a model-language with no formal data model-language. Because of this, the development of a parser for a model-language interface with no formal data model-language must be written in a lower-level form without the benefit of higher-level, program-development tools.

Another method of parser development is to design a KMS parser that uses the programming language C to effectively parse the data-definitions and request of the new model-language. Since much time will be spent developing the model-language representation, this approach is much easier than using higher-level programming tools such as Lex and Yacc [LESK 78] and [JOHN 78]. This affords the designers and programmers the opportunity to incorporate the design of the new model-language data structures into the KMS parsing process.

The third method is to write formal specifications for the new model-language. More specifically, we write the regular expressions for its legitemate tokens. We also write the BNF specifications for its syntatic structures and grammar. With these specifications, we can use LEX and YACC to generate the parser for KMS readily. We recommend this method for the development of KMS code.

3. Kernel Model-Language Mapping Strategies

In the case of a data model-language that has yet to be standardized, the development of a language to incorporate with the data model can be an involved process. We suggest the third method mentioned in the previous section. The overhead is of course the effort and time devoted to developing the formal specifications, both regular expressions and BNF specifications, for the new model-language.

By following our software methodology for the MM&MLDS, designers and programmers will be able to properly address considerations that must be made prior to the programming of the new model-language interface. Focusing on these issues prior to programming will save the design and program teams valuable man-hours developing a software methodology which in incompatible with MM&MLDS.

V. THE CONCLUSION

A. RESULTS OF OUR RESEARCH EFFORT

In this thesis, we have developed a paradigm for the introduction of new model-language interfaces on MDBS. By detailing a three-step process to a new model-language introduction into MDBS, a noticeable reduction in development time spent on new model-language interface incorporation will be realized. This paradigm will result in a more effective use of MDBS for both first-time user's and new model-language developers. It is therefore important for us to summarize the efforts of our research.

As a pedagogical aid for the instruction on the multimodel and multilingual database system, the MDBS User's Manual is the first step that designers must take in the process of introducing a new model-language interface. This manual has also proven instrumental in the educational environment as a supplement to the classroom lectures by providing students a vehicle with which to operate MDBS and thus further their understanding of advanced database topics.

The understanding of the procedures, methods, and tools required for introducing new model-languages into MM&MLDS is vital for designers of a new model-language. From our generic development algorithms for each software module of the Model/Language Interface, designers will be provided with a framework and foundation from which new model-languages will be implemented. The strict adherence to these algorithms will ensure smooth introduction of the new model-language into MM&MLDS. Our development of the generic data structure also provides the needed foundation from which the specifics of a new model-language may be implemented on MM&MLDS. Our explanation and outline of the relationship between the Test Interface and the Model/Language Interface sections of MM&MLDS allows designers to focus upon the interaction which must take place between the new model-language and the kernel database system. This is the heart of the MM&MLDS process.

Our software methodology outlines considerations that must be made when designing a new model-language interface for MDBS. Addressing key issues that will face designers of new model-languages, our methodology provides options for more effective model-interface design and incorporation.

We believe our paradigm is instrumental to the successful incorporation of new model-language interfaces into MDBS. The problems associated with heterogeneous databases are facing organizations world-wide and especially in the Department of Defense. The multibackend, multimodel, and multilingual approach to the database system design is an effective solution to these problems

B. FUTURE RESEARCH

The rapid growth of computer technology will have a tremendous impact upon MDBS. Conversion of the current Laboratory of Database System Research to the Sun4 RISC architecture will soon be realized. Research in the area of the software portability and backend configuration on this new architecture must be addressed.

With the successful incorporation of the Sun4 architecture, new tools such as X-Windows will be available for users and programmers. These new tools will be instrumental in the design of the new user interface for MDBS. Using a programming tool such as TAE Plus, a new user interface could be designed for MDBS, since TAE Plus generates code written in C and can thus be incorporated into MDBS software readily. The author has developed a prototype user interface for MDBS written in TAE Plus. The implementation of this interface would be an excellent research opportunity and allow developers of new model-languages to focus more on the structure of the new model-language by not having to develop a new interface.

APPENDIX A. MDBS USER'S MANUAL

LABORATORY FOR DATABASE SYSTEM RESEARCH
Naval Postgraduate School
Monterey, California

The Multibackend Database System (MDBS)

The Multimodel and Multilingual Database System User's Manual
Volume 1

by

Paul Alan Bourgeois

17 December 1992

MDBS Advisor: Dr. David K. Hsiao

Approved for public release; distribution is unlimited.

TABLE OF CONTENTS

I. INTRODUCTION 42
A. BACKGROUND 42
B. SCOPE 42
II. SYSTEM OVERVIEW 44
A. GENERAL 44
B. MULTIMODEL/MULTILINGUAL CAPABILITIES 44
C. MDBS LANGUAGE INTERFACE SOFTWARE MODULES 46
III. SCHEMA AND REQUEST FILES 48
A. OVERVIEW 48
B. SCHEMA FILES 48
C. REQUEST FILES 50
IV. GETTING STARTED AND RUNNING MDBS 53
A. MDBS PROCESSES 53
B. META-DISK MAINTENANCE 55
C. SETTING UP THE USER'S SCREEN 56
V. THE RELATIONAL/SQL INTERFACE 59
A. INTRODUCTION 59
B. LOADING A NEW DATABASE 60
C. PROCESSING AN EXISTING DATABASE 62
D. THE MASS LOAD FUNCTION 62
E. LOADING RECORDS USING SQL INSERT AND PROCESSING OTHER TRANSACTIONS 63
VI. THE NETWORK/CODASYL INTERFACE 67
A. INTRODUCTION 67
B. LOADING A NEW DATABASE 67
C. PROCESSING AN EXISTING DATABASE 70
D. LOADING AND EXECUTING CODASYL TRANSACTIONS VIA REQUEST FILES 71
VII. THE HIERARCHICAL/DL/1 INTERFACE 73
A. INTRODUCTION 73
B. LOADING A NEW DATABASE 74
C. PROCESSING AN EXISTING DATABASE 76
D. REQUEST FILE ORGANIZATION FOR LOADING DL/1 TRANSACTIONS 77
VIII. THE CROSS-MODELING ACCESS CAPABILITY 80
A. OVERVIEW 80
B. SCHEMA TRANSFORMATION 80

C. EXECUTING THE HIERARCHICAL TO RELATIONAL CROSS-MODELING CAPABILITY 81
IX. THE ATTRIBUTE-BASED DATA MODEL-LANGUAGE INTERFACE 84
A. OVERVIEW 84
B. DATABASE CONSTRUCTS 84
C. THE ATTRIBUTE-BASE DATA MODEL INTERFACE 85
1. The Template and Descriptor Files 85
2. The Mass Load File 87
3. Executing the ABDM interface 89
UM APPENDIX A: USEFUL UNIX AND MDBS COMMANDS 95
A. THE .alias FILE 95
B. C SHELLS 95
1. The zero C Shell 96
2. The stop.cmd C Shell 97
3. The run C Shell 98
C. THE README FILE 99
UM APPENDIX B: DEMO DATABASE EXECUTIONS 101
A. OVERVIEW 101
B. THE RELATIONAL/SQL INTERFACE 101
C. THE NETWORK/CODASYL INTERFACE 102
D. THE HIERARCHICAL/DL/1 INTERFACE 103
E. THE CROSS MODEL CAPABILITY 104
F. THE ATTRIBUTE-BASED/ABDL INTERFACE 104

I. INTRODUCTION

A. BACKGROUND

The Multi-backend Database System (MDBS) is the research database laboratory at NPS. Under the direction of Professor David K. Hsiao, this system provides a research testbed for solutions to the problems of heterogeneous databases and design concepts required for the development of a federated database system. The multiple data model and data language capability of MDBS allows the user to implement a wide variety of database models and data languages on a single system. Through its cross-model access capability, MDBS permits a user experienced in relational databases and the relational language SQL to access a hierarchical database by transforming the hierarchical schema into a relational schema and thus allowing transactions written in SQL to access the hierarchical database. Though a transformation of schema takes place, this transformation is transparent to the user.

B. SCOPE

The scope of this user's manual will be to explain all user interface aspects of the MDBS. Sections will include how to load and run a relational, hierarchical, and network database as well as how to use the cross-model capabilities of the system. Because the system uses the data sharing method of multiple data model/language mapping to a single (or kernel) data model/language, instructions are provided on the development and execution of an Attribute Based Database (ABD) using the Attribute Based Data Model (ABDM) and the Attribute Based Data Language (ABDL). Examples of certain data models and data languages are provided throughout this user's manual as well as formats used to construct the schema and request files necessary to build a database. A brief overview of the system is discussed in Chapter Two to familiarize the user with the hardware and software of MDBS. Appendix A provides a glossary of key terms and UNIX functions necessary for database implementation are also provided. Appendix B is the step-

by-step instructions required to load, process, and execute the four user interfaces on the MDBS.

II. SYSTEM OVERVIEW

A. GENERAL

The Multibackend Database System (MDBS) currently consists of six backend computers and one controller computer. Each backend is a database processor with its own micro-processor and database store. The backends are arranged in parallel in order to maximize performance through scalability, response-time reduction, and response-time invariance. Both paging and meta-data are stored are stored on separate 96 Megabyte Winchester type disk drives while the base data is maintained on much larger 500 Megabyte moving-head standard-type drives. The backends communicate via an Ethernet Local Area Network. The controller is different from the backends in that it does not require access to the meta-data and base data. The controller's main function is to provide backup and recovery of the backends. On the MDBS at the Naval Postgraduate School, DB8 (ISIV 8) with the front tape drive, is the controller. Figure 2-1 illustrates the relationship between the forward controller and the backend processors as well as the structure and organization of the MDBS. The number of backends the system can support is limited only by the number of connection slots provided for the backend processors by the controller.

B. MULTIMODEL/MULTILINGUAL CAPABILITIES

In order to support its claim of being multi-model/multi-lingual, MDBS supports four traditional data models and data languages as well as its own kernel data model and data language. MDBS supports these data models and data languages on the same system unlike traditional homogeneous database machines which support only one data model/ data language. The four data models/data languages supported by MDBS: relational/SQL, network/CODASYL, hierarchical/DL/1, and functional/DAPLEX (Author's note: at the time of this user's manual, the functional data model was in the process of being completed on MDBS), are all mapped to a kernel data model/data language on the MDBS system. It is this mapping process of the user's data model and data language into the kernel data model and data language that represent the heart of the MDBS.

The kernel data model used for MDBS is the attribute-based data model (ABDM) with its corresponding attribute-based data language (ABDL). The ABDM supports the five primary database operations: RETRIEVE, RETRIEVE COMMON, INSERT, UPDATE, and DELETE. Through clustering, the ABDM allows the base data to be partitioned into mutually exclusive sets and these sets of clusters to be distributed to the backends permitting parallel access to the base data.

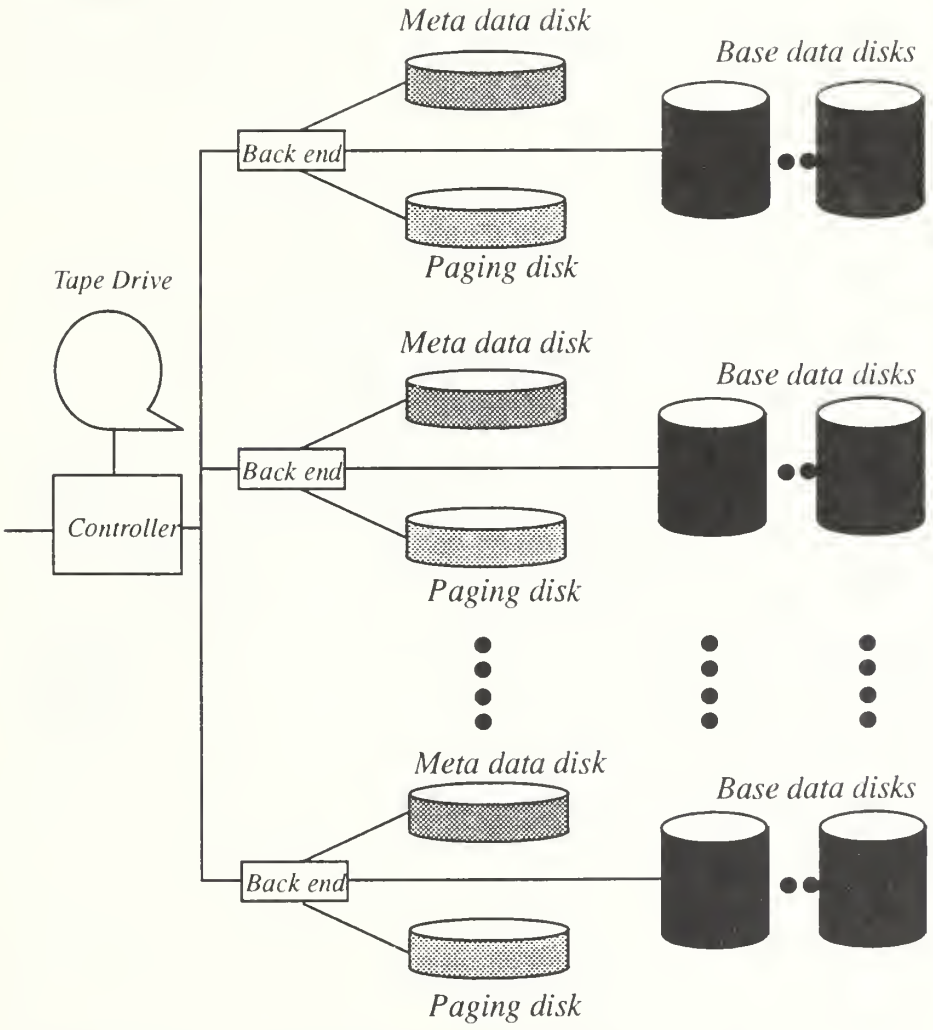


Figure 2-1
Structure and Organization of MDBS

C. MDBS LANGUAGE INTERFACE SOFTWARE MODULES

Each data model/data language supported by MDBS has four language interface software modules that perform the translation of the data model and data language into the kernel data model and data language. These four modules are the Language Interface Layer (LIL), Kernel Mapping System (KMS), Kernel Formatting System (KFS), and Kernel Controller (KC). A detailed description of each module as well as interface management strategies for loading new databases to the MDBS system, and a prototype for a new interface management system for MDBS can be found in [BOUR 93]. Figure 2-2 shows how each data model/data language is composed of the four software modules, with all data models linked to the Kernel Database System (KDS) and eventually to the Kernel Data Model (KDM) and Kernel Data Language (KDL).

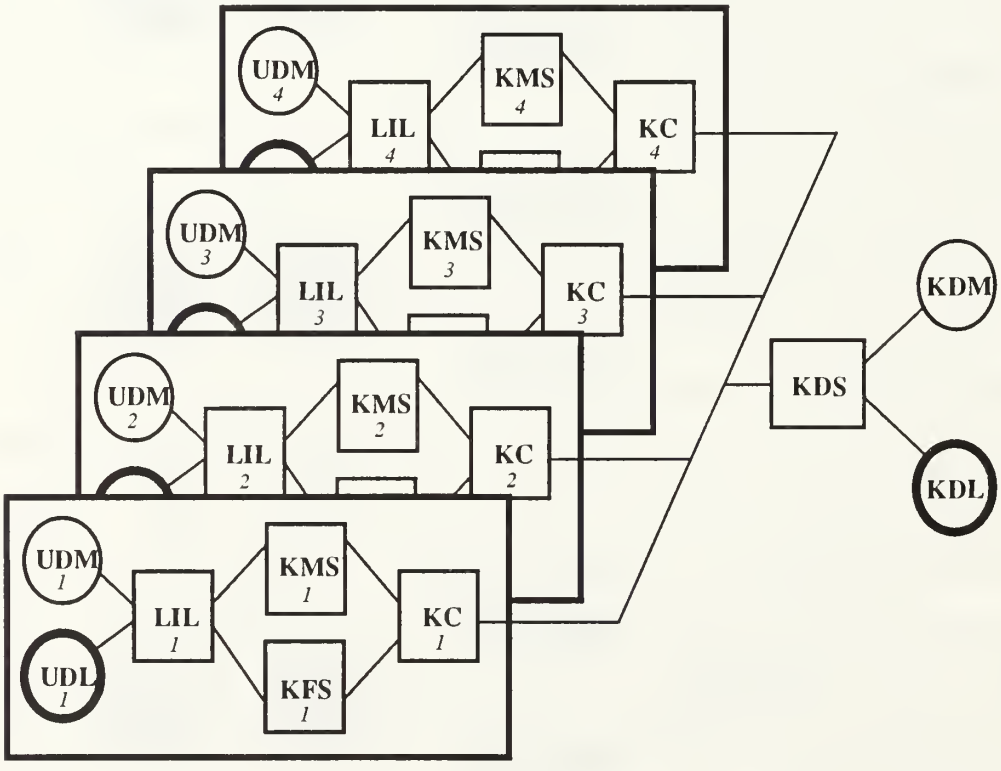


Figure 2-2
Multiple database interfaces linked to single kernel database system on MDBS

A demonstration/class account **cs4322** is set up to execute the MDBS from DB3. The password can be obtained from Professor Hsiao. An in-depth description of the hardware that comprises the MDBS system as well as MDBS performance studies and statistics can be found in [HSIA 91].

III. SCHEMA AND REQUEST FILES

A. OVERVIEW

Each of the databases developed on the MDBS system requires the use of a schema file and the option to use a request file. All schema and request files **MUST** be resident in the **UserFiles** directory in order to be used by MDBS since MDBS is hard - coded to check that directory to verify if the schema file actually exists. Files can be resident in a **subdirectory** of **UserFiles** but when the user is prompted by the system to input the schema or request file, the user **must** include the path name starting with the subdirectory name followed by a forward slash, and then the schema or request file name (Figure 3-1).

What is the name of the **CREATE/QUERY** file ----> /DEMO/COURSEsqldb

Figure 3-1
Request for SQL schema file

B. SCHEMA FILES

The schema file outlines the schema of the user's desired database in its respective query language, i.e. SQL, CODASYL, DL1, or ABDL. It is from the loading of this file that the descriptor and template (the ".d and .t" files) are generated by the Language Interface Layer.

For clarity, all schema files should be named in the following convention:

<database name><the acronym of the interface language (ie. sql, dml, dli)>db

For example, if a relational database name is COURSE, then its schema filename should be **COURSEsqldb**. There is no restriction to this rule, but once several databases have been developed, finding the corresponding schema file when executing a database can be lead to using the wrong schema file and having to start over. All schema files must have a dollar sign "$" on the last line of the file to mark the end of file. Otherwise, the parser will

not find and EOF mark and not process the schema file. Sample schema files are shown in Figure 3-2 through 3-4 and can also be found in the **cs4322** account in the **UserFiles/DEMO** directory.

```
create table employee: lname (char(8)),
                       fname (char(8)),
                       ssn (char(9)),
                       sex (char(1))
@
create table job: essn (char(9)),
                  position (char(10))
@
create table pay: essn (char(9)),
                  salary (int(6))
$
```

Figure 3-2
SQL schema file EMPRECsqldb

```
dbd   name= sqd
segm  name= co
field name= (sno, seq), bytes= 2
field name= sname, bytes= 2
segm  name= maint , parent= co
field name= (mdn, seq), type= int, bytes= 2
field name= mname, type= char, bytes= 2
segm  name= ops, parent= co
field name= (odn, seq, m), type= int, bytes= 2
field name= oname, type= char,  bytes= 2
segm  name= acft, parent= ops
field name= (ano, seq), type= int, bytes=2
field name= aname, type= char, bytes= 2
$
```

Figure 3-3
DL1 schema file SQDdlidb

```
SCHEMA NAME IS NET1;
RECORD NAME IS Supp;
  DUPLICATES ARE NOT ALLOWED FOR SNO;
    SNO ; CHARACTER 10.
    SNAME ; CHARACTER 10.
    CITY ; CHARACTER 10.
RECORD NAME IS Parts;
  DUPLICATES ARE NOT ALLOWED FOR PNO;
    PNO ; CHARACTER 10.
    PNAME ; CHARACTER 10.
    CITY ; CHARACTER 10.
RECORD NAME IS Purch;
    SNO ; CHARACTER 10.
    PNO ; CHARACTER 10.
    QTY ; FIXED 4.
SET NAME IS SuppPurc;
  OWNER IS Supp;
  MEMBER IS Purch;
  INSERTION IS AUTOMATIC
  RETENTION IS FIXED;
  SET SELECTION IS BY VALUE OF SNO IN Supp;
SET NAME IS PartPurc;
  OWNER IS Parts;
  MEMBER IS Purch;
  INSERTION IS AUTOMATIC
  RETENTION IS FIXED;
  SET SELECTION IS BY VALUE OF PNO IN Parts;
$
```

Figure 3-4
CODASYL schema file SQDdlidb

C. REQUEST FILES

The request file contains transactions the users wishes to process against a certain database. These transactions are written in the appropriate data model language given the data model of the database. The format for each request file is similar in that files which contain multiple transactions must have an "@" sign in between each transaction. All files must have an end of file marker (denoted by the dollar sign) on the last line of the file.

Sample request files are shown in Figures 3-5 through 3-7 and can also be found in the **cs4322** account in the **UserFiles/DEMO** directory.

```
select *
from employee
@
select *
from job
@
select *
from pay
@
select *
    from employee
    where sex = 'M'
@
select ssn,lname
    from employee
    where sex = 'F'
$
```

Figure 3-5
Figure Portion of SQL request file EMPRECreq

```
MOVE Sup1 TO SNO IN Supp
MOVE DEC TO SNAME IN Supp
MOVE MONT TO CITY IN Supp
STORE Supp
@
MOVE Part1 TO PNO IN Parts
MOVE NUT TO PNAME IN Parts
MOVE MONT TO CITY IN Parts
STORE Parts
@
MOVE Sup1 TO SNO IN Supp
FIND ANY Supp USING SNO IN Supp
GET Supp
$
```

Figure 3-6
Portion of CODASYL/DML request file NET1req

```
build (sno, sname) :
   ('A1', 'V2')
isrt co
@
build (mdn, mname) : (10, 'pr')
isrt co (sno = 'A1')
    maint
@
build (odn, oname) : (55, 'tp')
isrt co (sno = 'A2')
    ops
@
gu co
@
gu co (sno='A2')
  ops
@
gnp ops
$
```

Figure 3-7
Portion of DL1 request file
SQDreq

IV. GETTING STARTED AND RUNNING MDBS

Once all schema files and optional request files have been constructed, the user can now begin running the MDBS system. A user logging into the MDBS can use either the **mdbs** or the **cs4322** account. Both accounts will log into their respective default directory. The **mdbs** account is used primarily for thesis research and therefore has numerous directories from which the MDBS system may run from and options exists to predetermine the number of backends that the user wishes to use while running a particular database application. Due to constant manipulation and changes that occur from thesis research, our focus will be placed on using the **cs4322** account on the **db3** terminal. Appendix A covers key UNIX commands and C shells used to setup the MDBS.

Logging into db3 with the cs4322 account will take the user into the default directory of **db3/usr/work/cs4322**. The subdirectory **UserFiles** (or a subdirectory of **UserFiles**) should contain the schema and request file for the database to be processed. If this has not been done, the user must transfer his schema and request files into the **UserFiles** subdirectory (or a subdirectory of **UserFiles**).

Once all schema and request files are located in the proper directory, the user will change directories from the default path to the subdirectory **test.** Within the **test** directory, the following files exist: **README, run*, stop.cmd*, zero***. The **README** file outlines the limits for file name length, characters per attribute name as well as what the run*, stop.cmd*, zero* commands do. A copy of the **README** file is provided in Appendix A along with a listing of the run* stop.cmd* and zero* commands.

A. MDBS PROCESSES

Prior to executing the run command the user must verify that there are no processes still running the MDBS system. The UNIX command **ps ax** will display all active processes on your terminal whether you own those processes or not. Because an aborted run of the MDBS system can leave MDBS processes still running, the **ps ax** command will

help locate these processes and by using the UNIX command **kill**, you can kill those lingering processes. Look for any process like those highlighted in Figure 4-1:

```
PID TT STAT TIME COMMAND
 26590 ? IW 0:00 - desktop -d console (/etc/getty)
 26757 p0 I 0:02 rlogind
 26758 p0 IW 0:02 -csh (/bin/csh)
 26822 p0 IW 0:00 /bin/csh run
 26827 p0 I 0:00 /usr/work/cs4322/VerE.6/CNTRL/scntgpcl.out
 26828 p0 I 0:00 /usr/work/cs4322/VerE.6/BE/sbegpcl.out
 26829 p0 I 0:00 /usr/work/cs4322/VerE.6/CNTRL/scntppcl.out
 26830 p0 I 0:00 /usr/work/cs4322/VerE.6/CNTRL/pp.out
 26831 p0 I 0:00 /usr/work/cs4322/VerE.6/CNTRL/iig.out
 26832 p0 I 0:00 /usr/work/cs4322/VerE.6/CNTRL/reqprep.out
 26833 p0 I 0:00 /usr/work/cs4322/VerE.6/BE/dirman.out
 26834 p0 I 0:00 /usr/work/cs4322/VerE.6/BE/cc.out
 26835 p0 I 0:00 /usr/work/cs4322/VerE.6/BE/recproc.out
 26836 p0 IW 0:00 /usr/work/cs4322/VerE.6/BE/dio.out
 26837 p0 I 0:00 /usr/work/cs4322/VerE.6/BE/sbeppcl.out
 26839 p0 I 0:01 /usr/work/cs4322/VerE.6/CNTRL/dblti.out
 26844 p1 S 0:00 rlogind
 26845 p1 S 0:00 -csh (/bin/csh)
```

Figure 4-1
Result of executing the ps ax command

By typing **kill** and the process number, you can terminate the extraneous processes. A second method of kill the extraneous processes is to use the **stop.cmd** command. This command; the shell file is listed in Appendix A, will find all the extraneous processes running and kill them as shown in Figure 4-2. If the **stop.cmd** command is issued and no MDBS processes are running on the system, the user will be notified that there are no MDBS processes to kill as shown in Figure 4-3.

```
db3/usr/work/cs4322/test--38>stop.cmd
Stopping MBDS processes
killing 26827 26828 26829 26830 26831 26832 26833 26834 26835
26836 26839
```

Figure 4-2
Result of the stop.cmd command

```
db3/usr/work/cs4322/test--4> stop.cmd
Stopping MBDS processes
killing
kill: Too few arguments.
```

Figure 4-3
Executing the stop.cmd command with no MDBS processes running

B. META-DISK MAINTENANCE

Upon verification that no extraneous processes are running, the user must ensure that there is no existing database currently loaded to the system. This is accomplished by using the alias **pry**, this command will check the disk to make sure there is no data on it. The **pry** command will display what data is on the disk, if the line displays zeroes, as in Figure 4-4, then the data disk is clean and you are ready to execute the MDBS system.

```
0000000   \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
\0
*
```

Figure 4-4
Clean meta-disk

However, there may be an existing database stored on the disk, and the result of the **pry** command will look similar to Figure 4-5.

```
000000  \0 \0 003  E  M  P  R  E  C \0 \0 \0 \0 \0 \0 \0
0000016 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
```

Figure 4-5
Meta-disk with existing data

The **zero** command will let the user clean the meta-disk of any existing data. Users loading a new database to the system must ensure that the meta-disk is clean or the execution of the database will crash. Figure 4-6 displays what the user will see after executing the **zero** command.

Provided the user has either clean the meta-disk or plans to process an existing database, the user is now ready to execute the MDBS system. From the test directory, the user will type the command run to start the MDBS interface. Figure 4-7 illustrates what takes place after the run command is entered. (Author's note: The execution of the MDBS described in this manual is strictly for use with the class account cs4322. Execution of the MDBS using the thesis research account mdbs is accomplished in various ways with options to select the number of backends the user wishes to use. Since current thesis research is being conducted in the area of user selection of backends, a future appendixes appear in the next edition of the MDBS User's Manual, will outline the steps necessary to execute the MDBS using the thesis research account mdbs.).

C. SETTING UP THE USER'S SCREEN

It is advisable that the user use two C shells while operating the MDBS system. One shell will be used strictly for database execution while the other shell will be used for checking the **UserFiles** directory for ensuring that all necessary database files exist and so that the user can verify that all processes are running. When running the MDBS from the class account, the user will have six backend (BE) processes running and six control (CNTRL) processes running. These processes are highlighted in Figure 4-1. If the user does not have all these processes running, then the user must exit the system using **Control-c**, kill any extraneous processes with the **stop.cmd** command, double check to ensure no

extraneous processes are running using the **ps ax** command, ensure the data disk has been zeroed, and then restart the MDBS system using the **run** command

```
db3/usr/work/cs4322/test--
39>zero
No match.
No match.
File to zero = /dev/sd1c
File size = 105638400
Bytes to zero = 8000000
Bytes written...
   819200
  1638400
  2457600
  3276800
  4096000
  4915200
  5734400
  6553600
  7372800
  8000000
```

Figure 4-6
Result of the zero command

As seen in Figure 4-7, The Multi-Lingual/Multi-Backend Database System Menu offers an assortment of database models to choose. Since each interface has its own unique idiosyncrasies, the next section four sections will examine the Relational, Network, Hierarchical, and ABDM database models and respective user interfaces. Regardless of the database the user desires to implement, the commands and steps used to prepare and execute the MDBS pertain to all data models implemented on the MDBS system.

```
db3/usr/work/cs4322/test--42> run
[1] 29569
[2] 29570
[3] 29571
[4] 29572
[5] 29573
[6] 29574
[7] 29575
[8] 29576
[9] 29577
[10] 29578
[11] 29579

The Multi-Lingual/Multi-Backend Database System

Select an operation:

    (a) - Execute the attribute-based/ABDL interface
    (r) - Execute the relational/SQL interface
    (h) - Execute the hierarchical/DL/I interface
    (n) - Execute the network/CODASYL interface
    (f) - Execute the functional/DAPLEX interface
    (x) - Exit to the operating system

Select->
```

Figure 4-7
Result of executing the run command

V. THE RELATIONAL/SQL INTERFACE

A. INTRODUCTION

The relational interface on the MDBS system uses SQL as the data model language. Hence, all request files developed in conjunction with the relational interface will use SQL statements to manipulate the relational database. The schema file for the relational/SQL interface will resemble Figure 3-2 using the basic SQL constructs to create the schema file. The user must ensure the request and schema files are created prior to execution of the MDBS system.

At this point, the user has entered the run command and is looking at the Multi-Lingual/Multi-Backend Database System Menu as shown in Figure 4-7. Select **r** for the relational/SQL interface and the system will prompt the user for the operation desired (Figure 5-1). Select **option(l)** to load a new database, (**p**) to process a database currently

```
Enter type of operation desired
      (l)  - load new database
      (p) - process existing database
      (x) - return to the MLDS/MBDS system menu

Action --- >
```

Figure 5-1
System prompt to load a new or process an existing database

resident on the systems data disk, or (**x**) to return to the Multi-Lingual/Multi-Backend Database System Menu. After selecting **option (l)** or (**p**) the user will be prompted for the database name (Figure 5-2). If the user is loading a new database, the name of the database should (for clarity) give an idea of what the database represents (i.e. a database of class schedules could be called SCHEDULE). If the user desires to process an existing database, it is imperative that the database exist or the system will endlessly loop asking for the database name. If this situation presents itself, enter **Control-c** to abort the system and

start again as described in the **Getting Started** section. The database name may be written in all capital letters or small case letters, there is no restriction.

```
Enter name of database ---->
```

Figure 5-2
Enter database name prompt

B. LOADING A NEW DATABASE

Loading a new database differs from processing an existing database in that the new schema for the database has yet to be loaded to the system. After the user has entered the database name, the user will be prompted as to the mode of input desired to load the schema file (Figure 5-3). The user must either select **option (f)** and have a schema file already developed (this is highly recommended) or select the **option (t)** of loading the schema from the terminal. **Option (x)** will take the user back to the type of operation menu. Unscrewing instructions are provided if the user selects **option (t)**.

```
Enter mode of input desired
    (f) - read in a group of creates from a file
    (t) - read in creates from the terminal
    (x) - return to the main menu

Action --- >
```

Figure 5-3
Loading of the schema file

After selecting option (f) the user will be prompted to enter the name of the schema file. If the file resides in a subdirectory of UserFiles, the user must provide the path of the directory that the file resides as shown in Figure 5-4. As covered in the Schema and Request

File section of this manual, the schema file will create the template and descriptor files (the.t and .d files).

What is the name of the CREATE/QUERY file ----> DEMO/EMPRECsqldb

Figure 5-4
Enter schema file name prompt

The MDBS system will parse the schema file and transform the relational schema into the kernel data model language, ABDL. The parse will determine what the relations of the schema are and a screen will appear displaying the relations and a opportunity to index attributes in the relation if so desired. Figure 5-5 illustrates this action.

The following are the Relations in the **EMPREC** Database:

EMPLOYEE
JOB
PAY

Beginning with the first Relation, we will present each Attribute of the relation. You will be prompted as to whether you wish to include that Attribute as an Indexing Attribute, and, if so, whether it is to be indexed based on strict EQUALITY, or based on a RANGE OF VALUES. If you do not want to enter any indexes for your database, type an **'n'** when the **Action -->** prompt appears

Strike **RETURN** or **'n'** when ready to continue.
Action --- >

Figure 5-5
Relations from a relational schema file and attribute indexing

If the user desires to index attributes, onscreen instructions will direct the user how to properly index the attributes. The user will traverse through all relations and every attribute if indexing is desired. In most cases, the indexing is not used.

The user has now completed loading the new database schema onto the MDBS system. Once the user has either finished indexing the appropriate attributes or desired not to index attributes, the user will be prompted by Figure 5-1 and will select **option (p)** to process an existing database

C. PROCESSING AN EXISTING DATABASE

In order to process an existing database, the user will select **option (p)** in Figure 5-1 and will be prompted to enter the database name as in Figure 5-2. Since the database schema now exists on the MDBS system, the user may now input records into the system. After inputting the database name, the user will receive the screen display in Figure 5-6. **Options (f)**, **(t)**, and **(m)** all provide a means to input records into the database. The first two use SQL transactions to insert records into the database (as well as other transactions) while the third option allows the user to input several records into the database from a file. This option is the mass load function and it is only offered with the relational interface.

```
Enter mode of input desired
  (f)  - read in a group of queries from a file
  (t)  - read in queries from the terminal
  (m)  - mass load a file
  (d)  - display the current database schema
  (x)  - return to the previous menu

Action --- >
```

Figure 5-6
Record input menu

D. THE MASS LOAD FUNCTION

The mass load function is a unique method of loading records into each relation of the database. Without having to write several INSERT transactions in SQL, the user can load numerous records into the database using the mass load function. All mass load files contain the suffix ".r" and should be prefaced by the database name for clarity. After selecting the mass load option, the user will be prompted for the mass load file as in Figure

5-7. Figure 5-8 is an example of a mass load file. When developing a mass load file, the space in between attribute values along a tuple **must** be separated by a **TAB** and not the spacebar. The system will not read the space produce by the spacebar and assume one large attribute value or crash the system. As with the schema and request files, the mass load file must be maintained in the **UserFiles** directory or a subdirectory of **UserFiles**

Enter name of record file ----> DEMO/EMPREC.r

Figure 5-7
Prompt to input mass load file

After indicating the mass load file name, a series of ABDL insert statements will appear. If this does not happen then there has been an error in the mass load file and the system will crash. A period of inactivity greater than 20 seconds indicates problems with the mass load file and the necessity for the user to abort the system and restart. The mass load option does not replace the traditional SQL INSERT command but merely offers a faster method of initializing a database with base data. The traditional method of using SQL INSERT command to load the database is still a viable option on MDBS.

E. LOADING RECORDS USING SQL INSERT AND PROCESSING OTHER TRANSACTIONS

If the user desires to insert records into the relational database via the traditional method of using SQL INSERT commands and/or wishes to process SQL transactions against data currently residing on the database, **option (f)** will be chosen. The user must maintain a file within the **UserFiles** directory (or a subdirectory of **UserFiles**) that contains a list of the SQL transactions to be processed against the relational database. This file is commonly known as the request file and the file name is always contains the suffix "req". A prompt will appear after **option (f)** is selected requesting the user to input the name of the request file (Figure 5-9). Figure 3-5 shows an example of a request file developed for

the EMPREC database. An explanation of request file format is covered in the **Schema and Request File** section.

```
EMPREC
@
EMPLOYEE
SMITH     JOHN       111111111   M
JONES     BETTY      222222222   F
HART      PETE       333333333   M
THOMAS    TINA       444444444   F
JUDY      KEWIN      555555555   M
@
JOB
111111111 MANAGER
222222222 MANAGER
333333333 ACCOUNTANT
444444444 SECRETARY
@
PAY
111111111 50000
222222222 60000
333333333 45000
444444444 30000
$
```

Figure 5-8
Mass load file

```
What is the name of the CREATE/QUERY file ---->EMPRECreq
```

Figure 5-9
Prompt for request file

After inputting the request file, MDBS will scan the request file and, in the case of multiple transactions on one file, will number each transaction. If the display of the request file is longer than the screen display, the **-more-** prompt will be displayed in the bottom-left corner of the active window. Pressing the return key will display the remaining contents of the file. For longer files, this process may be repeated several times before reaching the

end of file. Once the transactions in the request file have been displayed to the user, a menu selection, as seen in Figure 5-10, will appear requesting the user to either execute a transaction, redisplay the contents of the request file, or exit back to the previous menu. Note that the first two options will execute and then return to the menu in Figure 5-10. If the user decides to use another request file or must enter a transaction via the terminal, entering **option (x)** will send the user to the menu in Figure 5-6

The user will process transactions against the database by simply entering the number of the transaction after the **Action** prompt in Figure 5-10. The transaction will be checked for semantic and syntactic correctness and, if in error, an appropriate message will be displayed. A serious error violating the conventions of the data language could result in a catastrophic error causing a bus error (core dump) and the need to restart the system.

```
Pick the number or letter of the action desired
(num) - execute one of the preceding queries
(d)   - redisplay the file of queries
(x)   - return to the previous menu

Action --- >
```

Figure 5-10
Transaction execution menu

If a transaction processes correctly, a display of the ABDL translation of the SQL transaction will be given and, in the case of a RETRIEVE operation, the data resulting from the query. This is illustrated in Figure 5-11.

Appendix B outlines the steps necessary to load and process the demonstration relational database EMPREC on the MDBS

```
[ RETRIEVE   (TEMP = Employee)(LNAME, FNAME, SSN, SEX)  ]

LNAME       |FNAME       |SSN             |SEX||
-----------------------------------------------
SMITH       |JOHN        |111111111       |M   |
JONES       |BETTY       |222222222       |F   |
HART        |PETE        |333333333       |M   |
THOMAS      |TINA        |444444444       |F   |
JUDY        |KEWIN       |555555555       |M
```

Figure 5-11
Result of a SQL retrieve operation with ABDL translation

VI. THE NETWORK/CODASYL INTERFACE

A. INTRODUCTION

The network interface on the MDBS system uses CODASYL as the data language for all network data models. All schema and request files require CODASYL statements in order to construct and manipulate the network databases on MDBS. The schema file for the network/CODASYL interface will resemble Figure 3-4 using the basic CODASYL constructs to create the schema file. The user must ensure that the request and schema files are created prior to execution of the MDBS system.

At this point, the user has entered the run command and is looking at the Multi-Lingual/Multi-Backend Database System Menu as shown in Figure 4-7. Select **n** for the network/CODASYL interface and the system will prompt the user for the operation desired (Figure 6-1). Select **option (l)** to load a new database, **option (p)** to process a database

```
Enter type of operation desired
        (l) - load new database
        (p) - process existing database
        (x) - return to the operating system

Action --- >
```

Figure 6-1
System prompt to load a new or process an existing database

currently resident on the systems data disk, or **option (x)** to return to the Multi-Lingual/ Multi-Backend Database System Menu. After selecting **option (l)** or **(p)** the user will be prompted for the database name (Figure 6-2). If the user is loading a new database, the name of the database should (for clarity) give an idea of what the database represents (i.e. a database of parts records could be called PARTS). If the user desires to process an existing database, it is imperative that the database exist or the system will endlessly loop asking for the database name. If this situation presents itself, enter Control-c to abort the system and start again as described in the **Getting Started** section. The database name

may be written in all capital letters or small case letters, there is no restriction.

```
Enter name of database ---->
```

Figure 6-2
Enter database name prompt

B. LOADING A NEW DATABASE

Loading a new database differs from processing an existing database in that the new schema for the database has yet to be loaded to the system. After the user has entered the database name, the user will be prompted as to the mode of input desired to load the schema file (Figure 6-3). Notice the difference between input options presented in the network interface and the relational interface (Figure 6-3). The network interface does not provide the option to input a schema from the terminal. Therefore it is imperative that the schema file exists prior to executing the network interface since further processing will not be possible without a schema file. Selecting **option (x)** will take the user back to the previous menu (Figure 6-1).

```
Enter mode of input desired
        (f) - read in database description from a file
        (x) - return to the to main menu

Action --- >
```

Figure 6-3
Loading the schema file

After selecting **option (f)** the user will be prompted to enter the name of the schema file. If the file resides in a subdirectory of **UserFiles**, the user must provide the path of the directory that the file resides as shown in Figure 6-4. As covered in the **Schema and Request File** section of this manual, the schema file will create the template and descriptor files (the.t and .d files).

What is the name of the DBD/REQUEST file ---->DEMO/NET1dmldb

Figure 6-4
Prompt to enter network schema file name

The MDBS system will parse the schema file and transform the network schema in the kernel data model, Attribute Based Data Model (ABDM), and kernel data language, Attribute Based Data Language (ABDL). The parse will determine the records that comprise the network database schema. The next screen will list the records of the network database as a result of reading the network schema file (Figure 6-5). Also available is the opportunity to index attributes for selected records in the database. If the user desires to

The following are the Records in the **NET1** Database:

SUPP
PARTS
PURCH

Beginning with the first Record, we will present each
Attribute of that Record. You will be prompted as to whether
you wish to include that Attribute as an Indexing Attribute,
and, if so, whether it is to be indexed based on strict
EQUALITY, or based on a RANGE OF VALUES. If you do not want
to enter any indexes for your database, type an '**n**' when
the **Action -->** prompt appears

Strike **RETURN** or '**n**' when ready to continue.
Action --- >

Figure 6-5
Records from a network schema and attribute indexing

index attributes, a prompt will appear for every attribute in every record as to whether index values for that attribute are desired. In most cases, indexing of attributes is not used.

The user has now completed loading a new network database schema onto the MDBS system. Once the user has either finished indexing the appropriate attributes or desired not

to index attributes, the user will be prompted by Figure 6-1 and will select **option (p)** to process an existing database.

C. PROCESSING AN EXISTING DATABASE

In order to process an existing database, the user will select **option (p)** in Figure 6-1 and will be prompted to enter the database name as in Figure 6-2. Now that the network database schema on the MDBS, the user may now execute CODASYL transactions against the database. Since there are no records stored in the database at this time (unless the schema had been loaded in a previous session and records were added during that session) the user should make the first group of transactions in the request file MOVE and STORE CODASYL transactions, in order to load the database with data prior to processing any query transactions.

The record input menu, Figure 6-6, will appear after inputting the correct database name. **Options (f)** and **(t)** provide the user the opportunity to either input transaction via a request file or through the terminal respectively. If **option (f)** is chosen, the user must unsure that the request file resides in the **UserFiles** directory or a subdirectory of **UserFiles.**

```
Enter mode of input desired
        (f) - read in a group of CODASYL requests from a file
        (t) - read in CODASYL requests from the terminal
        (d) - display the current database schema
        (x) - return to the previous menu

Action --- >
```

Figure 6-6
CODASYL record input menu

The name of the request file should use the name of the database (i.e. PARTS) as a prefix and "req" as a suffix, thus the name of the request file for the PARTS database would be PARTSreq. Additional request files can have an additional numerical suffix after "req" to identify how many request files exist for a database. Figure 3-6 illustrates a sample CODASYL request file. An explanation of the format used for request files is found in the

Schema and Request File section of this manual.

D. LOADING AND EXECUTING CODASYL TRANSACTIONS VIA REQUEST FILES

After select **option (f)**, the user will be prompted to enter the CODASYL request file name as in Figure 6-7. The user will enter the appropriate CODASYL request file name and a numbered list of transactions will appear on the screen. If the **-more-** message appears in the bottom lefthand corner of the screen, simply press **Return** to review the rest of the request file. After all transaction have been displayed, the user will be displayed a menu selection as shown in Figure 6-8.

```
What is the name of the DBD/REQUEST file ---->
```

Figure 6-7
Prompt for CODASYL request file

```
Pick the number or letter of the action desired
        (num) - execute one of the preceding CODASYL requests
        (d)   - redisplay the file of CODASYL requests
        (x)   - return to the previous menu

Action --- >
```

Figure 6-8
CODASYL transaction execution menu

The user may now begin executing CODASYL transactions against the database by entering the number corresponding to the transaction desired at the **Action --->** prompt in Figure 6-8. If a CODASYL STORE transaction is executed, the system will display the Attribute Based Data Language (ABDL) translation of the CODASYL request and then return to Figure 6-8 for further processing. A CODASYL GET transaction will generate an ABDL translation followed by the display of data from the query and then Figure 6-8 for further processing. If the user executes a CODASYL ERASE or MODIFY transaction, the

system will only display the ABDL translation of the CODASYL request and will return to Figure 6-8 upon completion. In order to ensure all STORE, ERASE, and MODIFY transaction accomplished what was desired, the user should include transactions in the request file that verify the correctness of these three transactions

Any errors that may result from violations of integrity constraints will be displayed to the user Catastrophic errors violating the conventions of the data language may result in the system crashing and restart of the system necessary. The user may simply follow the **option** (**x**) up to the Multi-Lingual/Multi-Backend Database System menu when finished with the CODASYL database. Appendix B provides a quick reference for executing the network/ CODASYL interface

VII. THE HIERARCHICAL/DL/1 INTERFACE

A. INTRODUCTION

The hierarchical interface on the MDBS uses DL/1 as the data language for all hierarchical data models. Schema and request files used in the hierarchical interface must use DL/1 statements in order to manipulate the hierarchical database on MDBS. Figure 3-3 is an example of a DL/1 schema file. As in the case of the network database, the DL/1 schema file must be created prior to execution of the MDBS since schema creation is not possible through terminal input.

The user has executed the **run** command from the UNIX prompt and should now have the Multi-Lingual/Multi-Backend Database System Menu, as shown in Figure 4-7, displayed. To begin the hierarchical/DL/1 interface, select **option (h)** and the desired operations menu will appear (Figure 7-1). **Option (x)** will return the user back to the MDBS menu. Whether the user selects **option (l)** or **(p)**, the next prompt will request the user to

```
Enter type of operation desired
        (l) - load new database
        (p) - process existing database
        (x) - return to the operating system

Action --- >
```

Figure 7-1
System prompt to load a new or process an existing database

input the database name as shown in Figure 7-2. If processing an existing database, the user should check the metadisk with the **pry** command prior to running MDBS to ensure that the database schema is resident on the system. If the user does not correctly input the name of an existing database, Figure 7-2 will infinitely loop until the user inputs an existing database name. **Control-c** will exit the user from the infinite loop and also the MDBS system. If this situation occurs, the user must kill all processes using the **stop.cmd** com-

mand and restart the system. There is no restriction as to the case of the letters comprising the database name, it is strictly the users preference.

```
Enter name of database ---->
```

Figure 7-2
MDBS prompt to enter database name

B. LOADING A NEW DATABASE

Loading a new database differs from processing an existing database in that the new schema for the database has yet to be loaded to the system. After the user has entered the database name, the user will be prompted as to the mode of input desired to load the schema file (Figure 7-3). Notice the difference between input options presented in the hierarchical interface and the relational interface (Figure 5-3). The network interface does not provide the option to input a schema from the terminal. Therefore it is imperative that the schema file exists prior to executing the network interface since further processing will not be possible without a schema file. Selecting option (x) will take the user back to the previous menu (Figure 7-1).

```
Enter mode of input desired
        (f) - read in database description from a file
        (x) - return to the to main menu

Action --- >
```

Figure 7-3
Loading the hierarchical schema file

After selecting **option (f)** the user will be prompted to enter the name of the schema file. If the file resides in a subdirectory of **UserFiles**, the user must provide the path of the directory that the file resides as shown in Figure 7-4. As covered in the **Schema and Request File** section of this manual, the schema file will create the template and descriptor files (the .t and .d files).

What is the name of the DBD/REQUEST file ---->DEMO/SQDdlidb

Figure 7-4
Entering the hierarchical schema filename

The MDBS system will parse the schema file and transform the hierarchical schema in the kernel data model, Attribute Based Data Model (ABDM), and kernel data language, Attribute Based Data Language (ABDL). The parse will determine the segments that compose the hierarchical database schema. The next screen will list the records of the hierarchical database as a result of reading the hierarchical schema file (Figure 7-5). As described in Figure 7-5, the user is given the opportunity to index specific attributes of each segment in the hierarchical database. Indexing is not a required function of MDBS and if the user does opt to use attribute indexing, all attributes in every segment will be screened for possible indexing.

The following are the Segments in the **SQD** Database:

CO
MAINT
OPS
ACFT

Beginning with the first Segment, we will present each
Field of that Segment. You will be prompted as to whether
you wish to include that Field as an Indexing Field,
and, if so, whether it is to be indexed based on strict
EQUALITY, or based on a RANGE OF VALUES. If you do not want
to enter any indexes for your database, type an **'n'** when
the **Action -->** prompt appears

Strike **RETURN** or **'n'** when ready to continue.
Action --- >

Figure 7-5
Segments from a hierarchical schema and attribute indexing

After completing the action in Figure 7-5, the user has now successfully loaded the hierarchical schema to the MDBS system and is ready to process DL/1 transactions against

the database. The user will be prompted by Figure 7-1 and will select **option (p)** to process an existing database. Figure 7-2 will appear prompting the user for the database name to process. Enter the name of the database whose schema was just loaded.

C. PROCESSING AN EXISTING DATABASE

In order to process an existing database, the user must ensure that the schema file has been loaded to the MDBS. If the schema file has not been loaded, select **option (l)** and load the schema file as described in the previous section. When processing an existing database, the user has the opportunity to load data into the database, execute queries, as well as modify and delete existing records. If the schema file has just been loaded to the system, the user should ensure the first group of transaction processed against the database are used to load initial data into the appropriate segments.

As with the other data models, the user will have the option of executing DL/1 transactions using a request file with transactions or creating transactions at the terminal as illustrated in Figure 7-6. If the user chooses **option (f)**, the request file should be named in the same convention as outlined in the other data mode sections (see Processing an Existing Database section for Network interface). A DL/1 request file is listed in Figure 3-3. If the user opts for terminal input of DL/1 transactions, onscreen instructions will explain how to properly enter and format the transactions.

```
Enter mode of input desired
        (f) - read in a group of DL/I requests from a file
        (t) - read in DL/I requests from the terminal
        (x) - return to the previous menu

Action --- >
```

Figure 7-6
Mode of input for DL/1 transactions

D. REQUEST FILE ORGANIZATION FOR LOADING DL/1 TRANSACTIONS

With the hierarchical database, any records loaded to the database must be done in a hierarchical order with those records belonging to the root segment loaded first, children

segments second and so forth. When creating your request file, it is easier to put the DL/1 BUILD commands at the beginning of the file and ordered in a hierarchical fashion in regards to the hierarchy of the segments.

After choosing **option (f)** or **(t)**, the user will be given a numerical listing on the screen of the DL/1 transactions either entered via a request file or through direct terminal input. If the **-more-** message appears in the bottom lefthand corner of the screen, simply press **Return** to review the rest of the request file. Note, after selecting **option (f)** the user will be prompt to input the request file name in the same manner as discussed on page 28 in the Network/CODASYL interface section and illustrated in Figure 6-7.

Upon completion of the transaction list, the user will be ready to execute those transactions from the menu in Figure 7-7. Unlike the relational and network interface, the hierarchical interface has a currency pointer which starts at the root segment. Any transaction that uses a DL/1 BUILD command must begin at the root segment in order to load data onto the database. Therefore, **option (r)** must precede the selection of **option (num)** for every BUILD transaction issued so that the data in the transaction can follow the

```
Pick the number or letter of the action desired
        (num) - execute one of the preceding DL/I requests
        (d)   - redisplay the file of DL/I requests
        (r)   - reset the currency pointer to the root
        (x)   - return to the previous menu

Action --- >
```

Figure 7-7
DL/1 transaction execution menu

path down the hierarchy to the segment where the data is to be stored. Figure 7-8 shows a sequence of resetting the currency pointer and executing a transaction. Note that a verification message is given upon any BUILD, IRST, and REPL transaction and all queries using GU, GHU, GNP will display the data requested and no message. All DL/1 transactions will have the equivalent ABDL translation displayed on screen after successful exe-

cution. The user can see from the ABDL translation how the transaction is traced through the database hierarchy in order to reach its destination segment.

After executing a GU transaction (Get Unique), if the next transaction is a GNP (Get Next Pointer) **within the same segment**, then the currency pointer does not need to be reset

```
Pick the number or letter of the action desired
    (num) - execute one of the preceding DL/I requests
    (d)   - redisplay the file of DL/I requests
    (r)   - reset the currency pointer to the root
    (x)   - return to the previous menu

Action --- > r

Pick the number or letter of the action desired
    (num) - execute one of the preceding DL/I requests
    (d)   - redisplay the file of DL/I requests
    (r)   - reset the currency pointer to the root
    (x)   - return to the previous menu

Action --- > 8

[ RETRIEVE  ((TEMP = Co) and (SNO = A2))  (SNO) BY SNO ]

[   RETRIEVE  ((TEMP = Ops) and (SNO = A2) and (ODN = 55))
(ODN) BY ODN ]

[   INSERT (<TEMP, Acft>, <SNO, A2>, <ODN, 55>, <ANO, 07>,
<ANAME, Bd>) ]

Insert accomplished
```

Figure 7-8
Sequence of selection when resetting currency pointer with ABDL translation and verification message. Note ABDL statements follow the hierarchy of the database in order to insert the data in the proper segment.

If the currency pointer is reset and the GNP transaction is executed, the error message in Figure 7-9 will be displayed. If an ISRT transaction is executed without resetting the currency pointer first, the user will be displayed the error message shown in Figure 7-10.

These examples are derived from the SQD hierarchical schema and request files found in the **DEMO** subdirectory of **UserFiles**. A quick reference for executing the hierarchical interface is provided in Appendix B. This quick reference will step the user through the demonstration hierarchical database SQD.

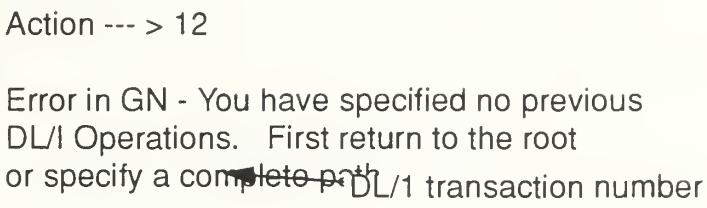

Action --- > 12
Error in GN - You have specified no previous
DL/I Operations. First return to the root
or specify a complete path
DL/1 transaction number

Figure 7-9
Error resulting when resetting the currency pointer is not required

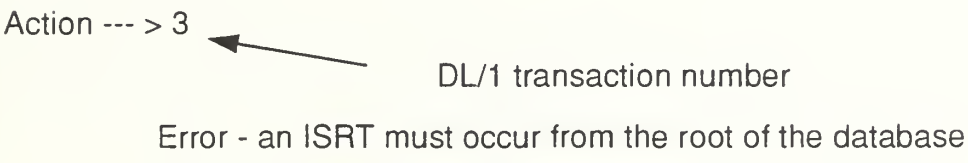


Figure 7-10
Failure to reset currency pointer for a DL/1 ISRT command

VIII. CROSS MODEL ACCESS CAPABILITY - RELATIONAL TO HIERARCHICAL

A. OVERVIEW

The benefit of MDBS as a multi-model/multi-lingual system has been shown throughout this manual. Large corporations using several separate homogeneous database to form one large heterogeneous database can benefit from the single system design of MDBS. With several different data models mapped to a kernel data model/data language on the same system, the sharing of data between two different data models is now possible as well as the vast amount saved through the consolidation of these several resources into one. MDBS offers the capability for a relational user to access a hierarchical database using SQL transactions. This is possible through the use of schema transformation and transaction translation. Instead of copying the existing hierarchical database into a relational database (and thus having to maintain two different databases), schema transformation permits the schema of the hierarchical database to be transformed into a relational schema so that SQL transactions may be processed

B. SCHEMA TRANSFORMATION

Figure 8-1 illustrates the concept of schema transformation of the hierarchical schema into a relational schema. The hierarchical database is transparent to the relational user who can now access the hierarchical database using the relational interface. SQL SELECT transactions map directly into the ABDL language and require no translations into the hierarchical data language. However, SQL INSERT, SELECT, and DELETE transactions require a translation into the hierarchical data language DL/1 in order to maintain the parent/child relationships of the hierarchical database. Because the relational schema cannot detect the parent/child relationships of the hierarchical schema, the SQL transactions cannot guarantee the parent/child relationship between segments will be violated. The transaction translator realizes that certain SQL transactions must be translated into DL/1 in order to maintain the integrity of the hierarchical database. The

work of the transaction translator and schema transformer is all transparent to the relational user who still can access the hierarchical database as though the database was strictly relational.

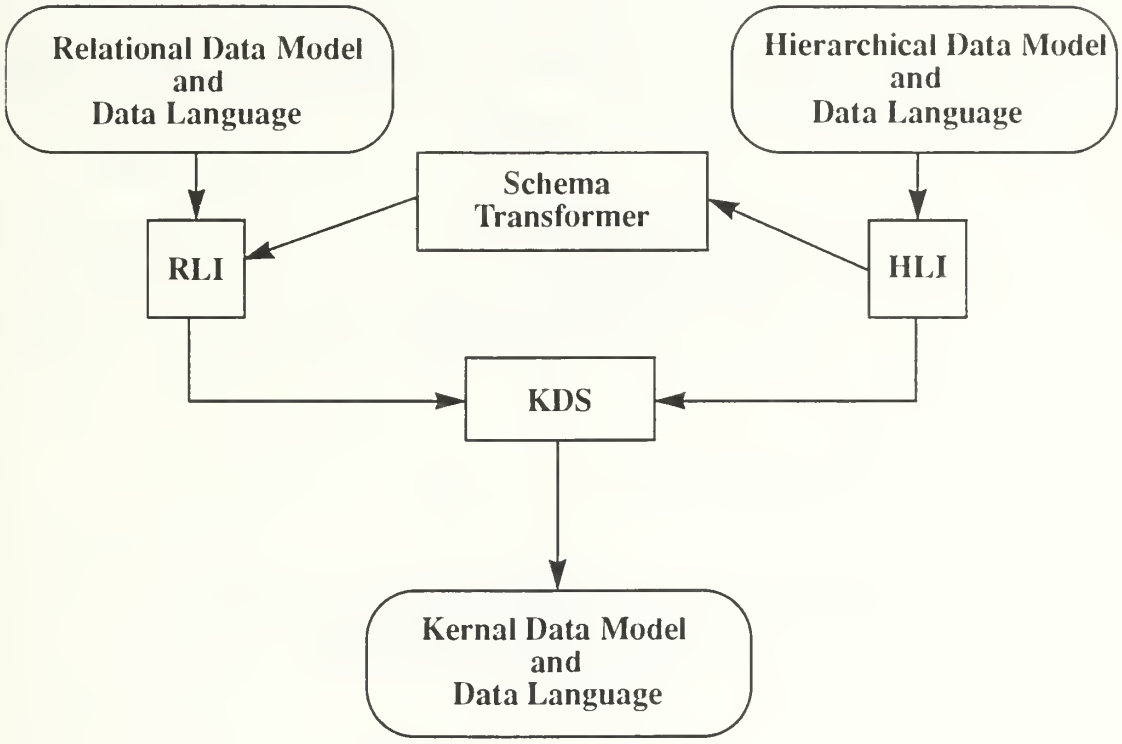


Figure 8-1
Schema Transformation Process

C. EXECUTING THE HIERARCHICAL TO RELATIONAL CROSS MODELING CAPABILITY

Now familiar with the concept behind the cross model capability of MDBS, the user is ready to use this capability. The first step the user must take is to load a hierarchical database as described in the **Hierarchical/DL/1 Interface** section. Only the schema has to be loaded to the hierarchical database. It is not necessary to load the hierarchical database with data using DL/1 transactions. Once the schema is loaded, the user should back out to

the Multi-Lingual/Multi-Backend Database System Menu and select **option (r)** for the relational/SQL interface The user will then request **option (p)** in Figure 8-2 and then enter the name of the hierarchical database just loaded.

The user will now be requested to select the mode of transaction input as shown in Figure 8-3. Even though the user is accessing and processing records against a hierarchical database, the user is given the same prompts as if the database was strictly relational. Transparency to the user is maintained at all times.

```
Enter type of operation desired
    (l) - load new database
    (p) - process existing database
    (x) - return to the MLDS/MBDS system menu

Action --- > p

Enter name of database ----> sqd
```

Figure 8-2
Processing an existing hierarchical database using the relational interface

```
Enter mode of input desired
(f) - read in a group of queries from a file
(t) - read in queries from the terminal
(m) - mass load a file
(d) - display the current database schema
(x) - return to the previous menu

Action --- >
```

Figure 8-3
Relational/SQL interface mode of transaction input screen

As with the other models, the user may choose a prepared file of SQL transactions for input or type transactions from the terminal. The mass load function will not function with

the cross-model capability and thus should not be chosen. The remaining steps mirror those covered in the **Relational/SQL Interface** section.

One important note: when the user is executing transactions within the relational/SQL interface, there is no need to reset the currency pointer as required when using the hierarchical/DL/1 interface. The schema transformation and transaction translation alleviates the relational user of having to maintain the currency pointer when executing SQL transaction. Currency pointer manipulation is transparent to the relational user.

A quick reference and an example of the execution of the cross-model access capability is provided in Appendix B. This appendix will use the hierarchical database SQD (also created in Appendix B) as the hierarchical database to be transformed into a relational schema.

IX. THE ATTRIBUTE-BASED DATA MODEL-LANGUAGE INTERFACE

A. OVERVIEW

The attribute-base data model (ABDM) is the kernel data model for the MDBS system. Coupled with the attribute-based data language (ABDL), this data model/data language is the key to the multiple data model/data language to single data model/data language mapping that makes MDBS a federated database. The ABDM was chosen as the kernel data model based because it stores the meta data and base data separately, introduces equivalence relations which partitions the base data into mutually exclusive sets called clusters, and allows these clusters to be distributed across the backends inducing parallel access to the base data.

ABDL was chosen as the kernel data language because it is a semantically rich and complete language such that transactions written in a traditional language like SQL, DL/1, or CODASYL, can be translated into ABDL. It is this translation capability that makes the MDBS mapping process a multiple data model/data language (i.e. relational/SQL, network/ CODASYL, hierarchical/DL/1) to a single data model/data language (i.e. ABDM/ABDL) mapping. ABDL also supports the five basic database operations of RETRIEVE, RETRIEVE COMMON, INSERT, MODIFY, and DELETE.

B. DATABASE CONSTRUCTS

Data in the ABDM is stored as an attribute-value pair. This attribute-value pair is the simple building blocks of the kernel database. The attribute-value pair consist of the attribute name and its corresponding value. When displayed, an attribute-value pair will be enclosed by a pair of angled brackets with the attribute name first, followed by the value for that attribute. An example would be <Vehicle, Car>, were Vehicle is the attribute name and Car would be its corresponding value.

A set of attribute-value pairs constitutes a record. Within a record, no two attribute-value pairs may have the same attribute-value name and at least one of the attributes in the

record is a key. These two rules ensure that each attribute-value pair is single valued and that the record can be identified by one attribute which is a key. A record is enclosed by parenthesis with attribute-value pairs within these parenthesis: (<Vehicle, Car>, <Manufacturer, Ford>, <Model, Explorer>, <Year, 1992>, <VIN, 1234567890>, <Owner, John Doe>).

A file is a collection of records that share unique set of attributes. If a record belongs to a certain file, then the first attribute-value pair of the record will contain the attribute File and the corresponding file name. All records belonging to the same file will have the same first attribute-value pair. For example, (<File, RegisteredCars>, <Vehicle, Car>, <Manufacturer, Ford>, <Model, Explorer>, <Year, 1992>, <VIN, 1234567890>, <Owner, John Doe>), would indicate that the record belonged to the file RegisteredCars.

and [HSIA 91] contain a detailed description of the ABDM and ABDL and the user is encourage to read these prior to executing the attributed-based interface.

C. THE ATTRIBUTE-BASE DATA MODEL INTERFACE

The user interface for the ABDM differs slightly from the three traditional data model interfaces discussed earlier. The ABDM interface does not require the use of a schema file or request file but instead uses template and descriptor files. In the three traditional data models, the schema file (with or without indexing) was used to generate the template and descriptor files necessary for mapping into the kernel data model/data language. In the ABDM interface the user must create the template and descriptor file prior to execution. A facility exists to generate a database but there are bugs in creating the descriptor file and therefore it is recommended that the user use a text editor (i.e. emacs or vi) to create the template and descriptor files.

1. The Template and Descriptor Files

The template and descriptor files (the .d and .t files) are used to describe the structure of the attribute-based database. It is these files which tell the kernel database system what the template names are and the attributes within a template. Furthermore, the

attribute type and any constraints on these attributes will be noted in these files. A template can be thought of as a name of relation in a relational database.

The template file contains the name of the database, followed by the number of templates within the database. After the number of templates, the next number is the number of attributes in the following template. The template name is listed followed by the attributes in that template and their respective type (i.e. string, integer, etc.). Once all attributes for a template are listed, the number of attributes in the next template is listed, followed by the next template's name. This process is repeated until all the templates and attributes have been listed. Figure 9-1 is an example of a template file for the demonstration database called SALES.

```
SALES
3
3
Item
TEMP s
PARTNO s
PARTNAME s
3
Customer
TEMP s
CUSTID s
CUSTNAME s
4
Order
TEMP s
CUSTID s
PARTNO s
PRICE i
```

Figure 9-1
Template File SALES.t

The descriptor file contains information with regards to constraints placed upon the attributes within the template. In order to achieve the mutual exclusivity of the MDBS, there are three descriptor types which an attribute may take on. *Type a* is an attribute which has a disjointed range of values (i.e. scores >0 <= 100). *Type b* is an attribute

of distinct value (i.e. color = black). *Type c* is an attribute that has a dynamic range that is determined at run time. In figure 9-2, attribute TEMP is a *type b* attribute whose distinct values are the template names in the database. The attribute PARTNO is a type a attribute whose value range is from P0001 to P9999. The attributes PARTNAME and CUSTNAME are also type a attributes whose value range goes from the letter A to Z.

```
SALES
TEMP b s
! Item
! Customer
! Order
@
PARTNO a s
P0001 P9999
@
PARTNAME a s
A F
G I
J P
Q T
U Z
@
CUSTNAME a s
A F
G I
J P
Q T
U Z
@
$
```

Figure 9-2
Descriptor File SALES.d

2. The Mass Load File

Like the relational/SQL interface, the ABDL interface also supports the mass load option to load records to the database. The mass load file will be named after the database name with a.r suffix. The mass load file format used in the ABDM interface is

exactly like the mass load file format used in the relational/SQL interface. In a sense, the mass load file resembles a template file with data instead of attribute names underneath the template name. The database name will appear at the top of the file followed by an @ symbol. After each template, an @ symbol must be used as a separator between templates. End of file is marked by a $ symbol. An important note when creating a mass load file, the system looks for TABS between attribute values in a record (or tuple). If the spacebar is used between attributes, the system will not read the space as the start of a new attribute and will erroneously read the mass load file. Figure 9-3 illustrates the mass load file for the demonstration database SALES.

```
SALES
@
Item
P8845 Widget
P3985 Bucket
P9002 Jack
P1233 Rake
P4501 Hammer
P4423 Ibeam
P7269 Vice
@
Customer
C101 Bradley
C102 Andrews
C103 Kreed
C104 Ridley
C105 Zaxxon
C106 Gibson
@
Order
C102 P9002 29
C103 P4501 19
C104 P8845 14
C106 P7269 79
C105 P4423 99
$
```

Figure 9-3
Mass Load File SALES.r

Once these files have been created, the user is ready to begin execution of the MDBS system using the ABDM interface. As with the previous data model interfaces, the user must ensure that the meta-disk has been cleared and that no extraneous MDBS processes are executing.

3. Executing the ABDM interface

After entering the run command from the test directory of the cs4322 account, select **option (a)** from The Multi-Lingual/Multi-Backend Database System menu like the one in Figure 4-7. The next menu to appear will look like Figure 9-4. Selecting **option (g)** will take the user through the steps needed to create template and descriptor files. Due to a bug in the option to create a descriptor file, this manual will only outline the steps required to create a template file through the ABDL interface.

```
The attribute-based/ABDL interface:

    (g)  - Generate a database
    (l)  -  Load a database
    (r)  - Request interface
    (x)  - Exit to the previous menu

Select->
```

Figure 9-4
ABDM interface menu

If the user selects **option (g),** a menu will appear similar to Figure 9-5 with a list of options in regards to file creation. Select **option (t) - Generate record template**. The user will then be prompted for the name of the template file. Remember to use the same name of the database and add the suffix.t. The user will then be prompted to enter the database ID; which is simply the database name. The ABDL interface is case sensitive to filenames similar to the manner in which UNIX is case sensitive. The user will then be lead through a series of questions in regards to the number of templates in the database and the

number of attributes within each template. Figure 9-5 shows the sequence of steps the user will have to go through using an example template file called TEST.t

```
Enter the template file name: TEST.t

ENTER DATABASE ID: TEST

ENTER THE NUMBER OF TEMPLATES FOR DATABASE TEST:
2

ENTER THE NUMBER OF ATTRIBUTES FOR TEMPLATE #1: 2

ENTER THE NAME OF TEMPLATE #1: Template1

ENTER ATTRIBUTE #1 FOR TEMPLATE Template1: ATT1A

ENTER VALUE TYPE (s = string, i = integer): s

ENTER ATTRIBUTE #2 FOR TEMPLATE Template1: ATT2A

ENTER VALUE TYPE (s = string, i = integer): s

ENTER THE NUMBER OF ATTRIBUTES FOR TEMPLATE #2: 3

ENTER THE NAME OF TEMPLATE #2: Template2

ENTER ATTRIBUTE #1 FOR TEMPLATE Template2: ATT1B

ENTER VALUE TYPE (s = string, i = integer): s

ENTER ATTRIBUTE #2 FOR TEMPLATE Template2: ATT2B

ENTER VALUE TYPE (s = string, i = integer): s

ENTER ATTRIBUTE #3 FOR TEMPLATE Template2: ATT3B

ENTER VALUE TYPE (s = string, i = integer): i
```

Figure 9-5
Sequence of steps in creating template file

Once the template file is completed exit the menu in Figure 9-4 and return to the attribute-based/ABDL interface menu, Figure 9-3. The user should have already created a descriptor file with the same name as the template file with a ".d" vice ".t" suffix. refer to Section 2 of this chapter if this has not been done.

The user is now ready to load the database to the MDBS. Select **option (l)** from Figure 9-3 and then choose **option (u)** from the next menu. A prompt will appear requesting the name of the database. Enter the name of the database (don't forget case sensitivity) and press return. The database template and descriptor files are now loaded to the meta-disk of the MDBS and the user is now ready to mass load data. Figure 9-6 illustrates this sequence of steps.

```
The attribute-based/ABDL interface:

    (g)  - Generate a database
    (l)  -  Load a database
    (r)  - Request interface
    (x)  - Exit to the previous menu

Select->
Select an operation:

    (u)  - Use a database
    (r)  - Mass load a file of records
    (x)  - Exit, return to previous menu

Select-> u

Enter the name of the database: SALES
```

Figure 9-6
Sequence steps to use an attribute-based database

The next menu to appear is a select operation menu similar to the second menu in Figure 9-6. Select **option (r) - Mass load a file of records**. A prompt will appear requesting the name of the mass load file. The mass load file must reside within the

UserFiles subdirectory of the cs4322 class account. Since the mass load file is loading data to the database, it may require a little more time to execute. A delay of up to 20 seconds is normal. If the mass load file was successful, the user will return to the previous menu and select **option (x)** to return to the attribute-based/ABDL interface menu. Figure 9-7 shows the screen display for the sequence of steps outlined above.

```
Select an operation:

        (u)  - Use a database
        (r)  - Mass load a file of records
        (x)  - Exit, return to previous menu

Select-> r

Enter the record file name: SALES.r

Select an operation:

        (u)  - Use a database
        (r)  - Mass load a file of records
        (x)  - Exit, return to previous menu

Select-> x

The attribute-based/ABDL interface:

        (g)  - Generate a database
        (l)  - Load a database
        (r)  - Request interface
        (x)  - Exit to the previous menu

Select->
```

Figure 9-7
Steps in processing the mass load file

At the attribute-based/ABDL interface menu, select **option (r) - Request interface**. The next menu to appear will be the subsession menu that appears in Figure 9-

8. The beginning user should only be concerned with **option** (s), (n), and (d) of the subsession menu. **Option (p)** is used primarily for setting the internal and external clock to gauge system performance when increasing/decreasing the number of backends. **Option (r)** allows the user to choose where the output from the ABDL transactions should be routed. The choice of routes are printer, CRT (or monitor), both, or none. **Options (m)** and (o) are self-explanatory with **option (m)** being a menu driven process of adding, modifying, or removing traffic units from a file. Note: traffic units is the name given for ABDL transactions. A list of traffic units is normally stored in a file similar to a request file in the other three data model interfaces and is sometimes referred to as a traffic unit file.

```
Select a subsession:
    (s)  SELECT: select traffic units from an existing list
         (or give new traffic units) for execution
    (n)  NEW LIST: create a new list of traffic units
    (d)  NEW DATABASE: choose a new database
    (p)  * PERFORMANCE TESTING
    (r)  * REDIRECT OUTPUT: select output for answers
    (m)  * MODIFY: modify an existing list of traffic units
    (o)  * OLD LIST: execute all the traffic units in an
         existing list
    (x)  EXIT: return to previous menu

Refer to the MLDS/MBDS user manual before choosing
subsessions marked with an asterisk (*)
Select-->
```

Figure 9-8
Subsession menu

Option (n) allows the user to create a new list (or file) of traffic units to process against an existing database. This is also menu driven. **Option (n)** lets the user switch to a new database (provided template and descriptor files exist). **Option (s)** is were the user may execute traffic units against the attribute-based database. Select **option (s)** and

a prompt will appear requesting the name of the traffic unit file to be processed. Note: the traffic unit file must contain a '#' symbol before the traffic unit file version number. For example, the database SALES has a first version traffic unit file called SALES#1, the second version traffic unit file is SALES#2, and so forth. After the traffic unit file is entered, the screen will display a numbered list of the traffic units to be processed. A menu will appear, see Figure 9-9, prompting the user to execute a traffic unit, create a new traffic unit, redisplay traffic units, or return to subsession menu.

```
Select Options:

    (d)     redisplay the traffic units in the list
    (n)     enter a new traffic unit to be executed
    (num)   execute the traffic unit at [num]
            from the above list
    (x)     exit from this SELECT subsession

Option ->
```

Figure 9-9
Traffic unit processing menu

At the Option-> prompt, enter the number corresponding to the traffic unit to be processed. After all traffic units have been processed the user may exit the MDBS, build a new attribute-based database, or process a new traffic unit file against the original database. Appendix B provides a step-by-step outline of executing the demonstration attribute-based database SALES.

UM APPENDIX A: USEFUL UNIX AND MDBS COMMANDS

A. THE .ALIAS FILE (ABRIDGE)

TABLE 1: Common System Commands Used Prior to MDBS Execution

ALIAS	UNIX COMMAND	EXPLANATION
p	ps x	shows all running processes.
test	cd ~/test	change directory to test.
x	exit	logs user out of db3.
user	cd ~/UserFiles	change directory to UserFiles.
data	cd ~/UserFiles	change directory to UserFiles.
disk	cd ~/RunData/Disks	change directory to Disks. Base data is stored here.
pry	od -c /dev/sd1c	octal display of the meta-disk. Zeroes indicate meta-disk is clean.
burn	~/test/stop.cmd	stop all current MDBS processes.

These aliases are the most commonly used on the MDBS. In the file .alias in the cs4322 default directory, a list of more aliases can be found. These aliases deal primarily with accessing the directories which contain the code for the user interface as well as the backend software.

B. C SHELLS

MDBS uses three C shells to ready the system for execution. The burden of remembering several UNIX commands in a specific order is no longer placed upon the user when the C shells are used. These C shells are found in the **test** directory of the cs4322 account and are zero*, stop.cmd, and run*. Figures A-1 through A-3 list the three C shells used by the MDBS software and a brief explanation of each C shell function is also provided.

1. The zero C Shell

The zero C shell will clean the meta-disk of all previous data. After executing the zero command, the user may be echoed with "No Match". This is a result of the remove file command not finding the file it was supposed to remove. Most of these UNIX *awk* commands are issued in case the previous execution of the MDBS resulted in abnormal termination and these files were never removed from the system. The **"cp /dev/null ~/RunData/Disks/disk0"** line will copy null bytes into the file were the base data is stored (the disk0 file). The last line ,**"/usr/work/cs4322/.z /dev/sd1c 8000000"**, will specify the number of bytes that will be copied over. In Figure A-1, 8000000 bytes are to be nulled.

```
#file: /usr/work/cs4322/test/zero
rm -f ~/test/.*cat
rm -f ~/test/.sql.dbl
rm -f ~/test/.hie.dbl
rm -f ~/UserFiles/.R*
rm -f ~/UserFiles/*.iig.at
rm -f ~/UserFiles/*.cinbt
cp /dev/null ~/RunData/Disks/disk0
/usr/work/cs4322/.z /dev/sd1c 8000000
```

Figure A-1
The zero* C shell

2. The stop.cmd C Shell

The stop.cmd C shell, Figure A-2, kills any MDBS process running on the system. The process numbers killed will be listed across the screen. The file .test.awk contains the parameters by which the UNIX *awk* command will search. If any parameters are found, they are copied to a file list2.stop. The list2.stop file contains the process numbers of the jobs to be killed. After the jobs are killed, the list2.stop file is erased as well as any data existing in the G_CPCLC and G_G_BPCLB sockets of the RunData directory.

```
# file:  /usr/work/cs4322/test/stop.cmd
echo 'Stopping MBDS processes'
ps -x | awk -f .test.awk >! list2.stop

set noclob
set a=`cat list2.stop`
echo killing $a
      kill $a
#
rm list2.stop
rm -f /usr/work/cs4322/RunData/G_BPCLB
rm -f /usr/work/cs4322/RunData/G_CPCLC
##rm -f *.tr core .b*
```

Figure A-2
The stop.cmd C shell

3. The run C Shell

The run C shell, Figure A-3, starts the MDBS system by calling all the executable files that drive the backends and controller. Tracer files can be used to track places were the system failed during a particular session. The commented lines (those pre-

```
# file: /usr/work/cs4322/test/run
#
rm -f /usr/work/cs4322/RunData/G_BPCLB
rm -f /usr/work/cs4322/RunData/G_CPCLC

/usr/work/cs4322/VerE.6/CNTRL/scntgpcl.out >&! /dev/null &
/usr/work/cs4322/VerE.6/BE/sbegpcl.out >&! /dev/null &

/usr/work/cs4322/VerE.6/CNTRL/scntppcl.out >&! /dev/null &
/usr/work/cs4322/VerE.6/CNTRL/pp.out >&! /dev/null &
#/usr/work/cs4322/VerE.6/CNTRL/pp.out >&! /usr/work/cs4322/test/pp.tr &
/usr/work/cs4322/VerE.6/CNTRL/iig.out >&! /dev/null &
/usr/work/cs4322/VerE.6/CNTRL/reqprep.out >&! /dev/null &
#/usr/work/cs4322/VerE.6/CNTRL/reqprep.out >&! /usr/work/cs4322/test/reqp.tr &
#
/usr/work/cs4322/VerE.6/BE/dirman.out >&! /dev/null &
#/usr/work/cs4322/VerE.6/BE/dirman.out >&! /usr/work/cs4322/test/dm.tr &
/usr/work/cs4322/VerE.6/BE/cc.out >&! /dev/null &
/usr/work/cs4322/VerE.6/BE/recproc.out >&! /dev/null &
#/usr/work/cs4322/VerE.6/BE/recproc.out >&! /usr/work/cs4322/test/recp.tr &
/usr/work/cs4322/VerE.6/BE/dio.out >&! /dev/null &

# must NOT start until cntgpcl is listening
(sleep 10 ; \
      /usr/work/cs4322/VerE.6/BE/sbeppcl.out >&! /dev/null ) &

/usr/work/cs4322/VerE.6/CNTRL/dblti.out 1

#chmod g+w *.tr
#rm *.tr core .b*
```

Figure A-3
The run C shell

ceded by a #) that are used in calling an executable file, are setup to use tracer files. If the user desires to use tracer files, then the executable files set up with a tracer file (i.e. #/usr/work/cs4322/VerE.6/CNTRL/pp.out >&! /usr/work/cs4322/test/pp.tr &) must comment out the matching line containing the same executable file name which sends tracer information to null (i.e. /usr/work/cs4322/VerE.6/CNTRL/pp.out >&! /dev/null &).

C. README FILE

The README file can be found in the test directory of the **cs4322** account. As shown in Figure A-4, the README file contains a list of system constraints place upon file size, attribute length case sensitivity, etc. The user should always have a copy of the README file when developing a new database in order to have a quick reference to the system constraints.

```
                    LIMITS OF YOUR SINGLE USER SYSTEM
                    ---------------------------------------------

max length of unix file name for the template & descriptor file is
      10 chars.

max retrieves per transaction is 5

max chars for attribute name is 10

max chars for attribute value is 30

max # of attributes you can have per template is 50

max # of descriptors you can have is 150 per attribute

max # of clusters you can have is 100

max # of records per cluster is 200

max # of templates per db is 10

max # of traffic units per query file is 25

the largest integer value you can use to join on a retrieve
common is 2147483647.
```

Figure A-4
The README file

UM APPENDIX B: EXECUTION OF DEMONSTRATION DATABASES

A. OVERVIEW

The purpose of this appendix is to provide the user with step-by-step reference to executing the demo hierarchical, relational, network, and cross model databases that are stored in the **DEMO** directory of the **UserFiles** directory in the **cs4322** class account. These step-by-step instructions will aid the user in developing new databases on the MDBS by familiarizing the user with the MDBS system and giving the user a hands-on opportunity to work with the MDBS interface.

Prior to executing the interfaces, the user must ensure that there are no extraneous MDBS processes running (use the UNIX **ps ax** command to verify) and that the meta-disk is clear (using the **pry** command). Note: the meta-disk must have data from the hierarchical database in order to execute the cross-model accessing capability.

B. THE RELATIONAL/SQL INTERFACE

- Type **run** from the **mdb3/usr/work/cs4322/test** directory.
- From the MDBS menu select **option (r) - Execute the relational/SQL interface.**
- Select **option (l) - load new database** from the type of operation menu.
- Enter the database name **EMPREC** (caps not required).
- Select **option (f) - read in a group of creates from a file** from the mode of input desirec menu.
- At the **What is the name of the CREATE/QUERY file** prompt, enter the schema file name **DEMO/EMPRECsqldb**.
- If prompted to use the existing descriptor file, **EMPREC.d**, enter **n** for no.
- After the relations of the database are displayed, enter **n** for no indexing.
- Select **option (p) - process existing database** for the mode of operation.
- Enter the database name **EMPREC** (caps not required).
- For mode of input desired, enter **option (m) - mass load a file**.
- Enter **DEMO/EMPREC.r** at the **name of record file--->** prompt
- After the records from the mass load have been displayed, select **option (f) - read in a group of queries from a file** at the mode of input menu.

- At the **What is the name of the CREATE/QUERY file** prompt, enter the request file name **DEMO/EMPRECreq**.
- The SQL transactions in the request file **EMPRECreq** will be displayed and numbered. Hit the **return** key if the **-more-** prompt is displayed in the bottom right hand corner in order to finish scrolling through the SQL transactions.
- At the action prompt enter the number of the SQL transaction you wish to process. Enter 1, let the transaction process and observe the results, then execute transaction 2 and so forth until all the transactions in the request file have been processed.
- Once completed, enter **option (x) - return to the previous menu** at the **Pick the number or letter of the action desired** menu.
- Keep selecting **option (x)** until you have exited the MDBS system.
- Type **zero** from the **mdb3/usr/work/cs4322/test** to clean the meta-disk in order to execute the next interface.

C. THE NETWORK/CODASYL INTERFACE

- Type **run** from the **mdb3/usr/work/cs4322/test** directory.
- From the MDBS menu select **option (n) - Execute the network/CODASYL interface**.
- Select **option (l) - load new database** from the type of operation menu.
- Enter the database name **NET1** (caps not required).
- Select **option (f) - read in a group of creates from a file** from the mode of input desirec menu.
- At the **What is the name of the DBD/REQUEST file** prompt, enter the schema file name **DEMO/NET1dmldb**.
- If prompted to use the existing descriptor file, **NET1.d**, enter **n** for no.
- After the records of the database are displayed, enter **n** for no indexing.
- Select **option (p) - process existing database** for the mode of operation.
- Enter the database name **NET1** (caps not required).
- Select **option (f) - read in a group of queries from a file** at the mode of input menu.
- At the **What is the name of the DBD/REQUEST file** prompt, enter the request file name **DEMO/NET1req**
- The CODASYL transactions in the request file **NET1req** will be displayed and numbered. Hit the return key if the **-more-** prompt is displayed in the bottom right hand corner in order to finish scrolling through the CODASYL transactions.
- At the action prompt enter the number of the CODASYL transaction you wish to process. Enter 1, let the transaction process and observe the results, then execute transaction 2 and so forth until all the transactions in the request file have been processed.

- Once completed, enter **option (x) - return to the previous menu** at the Pick the number or letter of the action desired menu.
- Keep selecting **option (x)** until you have exited the MDBS system.
- Type **zero** from the **mdb3/usr/work/cs4322/test** to clean the meta-disk in order to execute the next interface.

D. THE HIERARCHICAL/DL/1 INTERFACE

- Type **run** from the **mdb3/usr/work/cs4322/test** directory.
- From the MDBS menu select **option (h) - Execute the hierarchical/DL/I interface**.
- Select **option (l) - load new database** from the type of operation menu.
- Enter the database name **SQD** (caps not required).
- Select **option (f) - read in a group of creates from a file** from the mode of input desirec menu.
- At the **What is the name of the DBD/REQUEST file** prompt, enter the schema file name **DEMO/SQDdlidb**.
- If prompted to use the existing descriptor file, **SQD.d**, enter **n** for no.
- After the segments of the database are displayed, enter **n** for no indexing.
- Select **option (p) - process existing database** for the mode of operation.
- Enter the database name **SQD** (caps not required).
- Select **option (f) - read in a group of queries from a file** at the mode of input menu.
- At the **What is the name of the DBD/REQUEST file** prompt, enter the request file name **DEMO/SQDreq**
- The DL/1 transactions in the request file **SQDreq** will be displayed and numbered. Hit the return key if the **-more-** prompt is displayed in the bottom right hand corner in order to finish scrolling through the DL/1 transactions.
- In order to execute the **first ten DL/1 transactions**, the user must first select **option(r) - reset the currency pointer to the root**, then the transaction number. Since the irst transactions that are being loaded to the hierarchical database must be in a hierarchical order (i.e. data loaded to the root segment first, then children segments, etc.), the root pointer must be reset after each transaction to ensure a complete path from the root to the children segments.
- Reset the currency pointer prior to **transaction 11** but do not reset the currency pointer for **transaction 12.** This is because the currency pointer is already at the seqment from which the GET operation is being executed.
- Reset the currency pointer prior to **transaction 13** and then execute **transaction 14** since the currency pointer does not need to be reset prior to the execution of transaction 14.
- The currency pointer must be reset prior to executing both **transactions 15 and 16**.

- Once completed, enter **option (x) - return to the previous menu** at the Pick the
- number or letter of the action desired menu.
- Keep selecting **option (x)** till you reach **The Multi-Lingual/Multi-Backend Database System** menu. **Do not exit the MDBS system if you wish to use the cross model capability of the MDBS.** Go to Section E for the steps need to execute the cross model interface.

E. THE CROSS MODEL CAPABILITY

- Without exiting the MDBS system, the user should have the hierarchical database SQD loaded to MDBS and be at **The Multi-Lingual/Multi-Backend Database System** menu.
- From the MDBS menu select **option (r) - Execute the relational/SQL interface.**
- Select **option (p) - process existing database** for the mode of operation.
- Enter the database name **SQD** (caps not required).
- Select **option (f) - read in a group of queries from a file** at the mode of input menu.
- At the **What is the name of the CREATE/QUERY file** prompt, enter the request file name **DEMO/SQDRTHreq**. This file contains SQL transactions to be processed against a hierarchical database already loaded to MDBS.
- The SQL transactions in the request file **SQDRTHreq** will be displayed and numbered. Hit the return key if the **-more-** prompt is displayed in the bottom right hand corner in order to finish scrolling through the SQL transactions.
- At the action prompt enter the number of the SQL transaction you wish to process. Enter 1, let the transaction process and observe the results, then execute transaction 2 and so forth until all the transactions in the request file have been processed. Notice that there is no longer a need for a currency pointer as in the hierarchical database interface.
- Once completed, enter **option (x) - return to the previous menu** at the Pick the number or letter of the action desired menu.
- Keep selecting **option (x)** until you have exited the MDBS system.
- Type **zero** from the **mdb3/usr/work/cs4322/test** directory to clean the meta-disk in order to execute the next interface.

F. THE ATTRIBUTE-BASED/ABDL INTERFACE

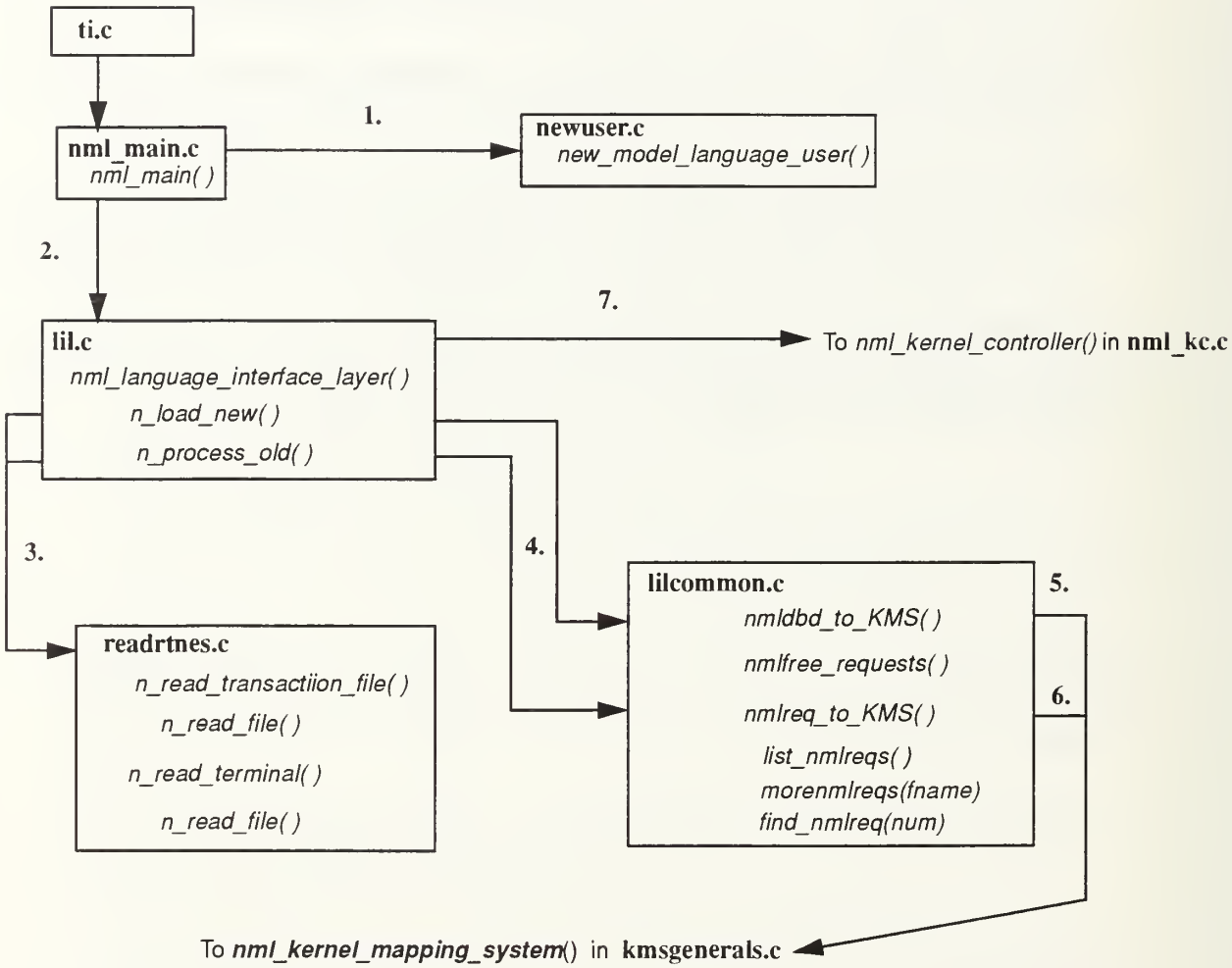
- From the MDBS menu select **option (a) - Execute the attribute-based/ABDL interface.**
- Select **option (l) - Load a database** from the attribute-based/ABDL interface menu..
- Select **option (u) - Use a database.**
- Enter **SALES** as the name of the database name of the database to be processed.

Remember that the ABDL interface is case sensitive. Also, there must be template and descriptor files already created for the database to be processes. If not, exit and create a new template file using **option (g) - Generate a database**. The descriptor file must be created using a test editor like emacs or vi.

- After entering the database name, select **option (m) - Mass load a file of records**. Enter **SALES.r** as the mass load file to be processed.
- Select **option (x)** to return to the attribute-based/ABDL interface menu.
- Select **option (r) - Request interface**.
- From the subsession menu, select **option (s) SELECT: select traffic units from an existing list (or give new traffic units) for execution**.
- Enter the filename **SALES#1** as the traffic unit file to be processed.
- After entering the traffic unit filename, a numbered list of all traffic units in the traffic unit file will be displayed. From the menu, select the number of the traffic unit to be processed. There are only 11 traffic units to be executed.
- After all traffic units have processed, exit the MDBS system by selecting **option (x)** at all menu that appear. Be sure to clean the meta-data disk after exitting the system by using the **zero** command.

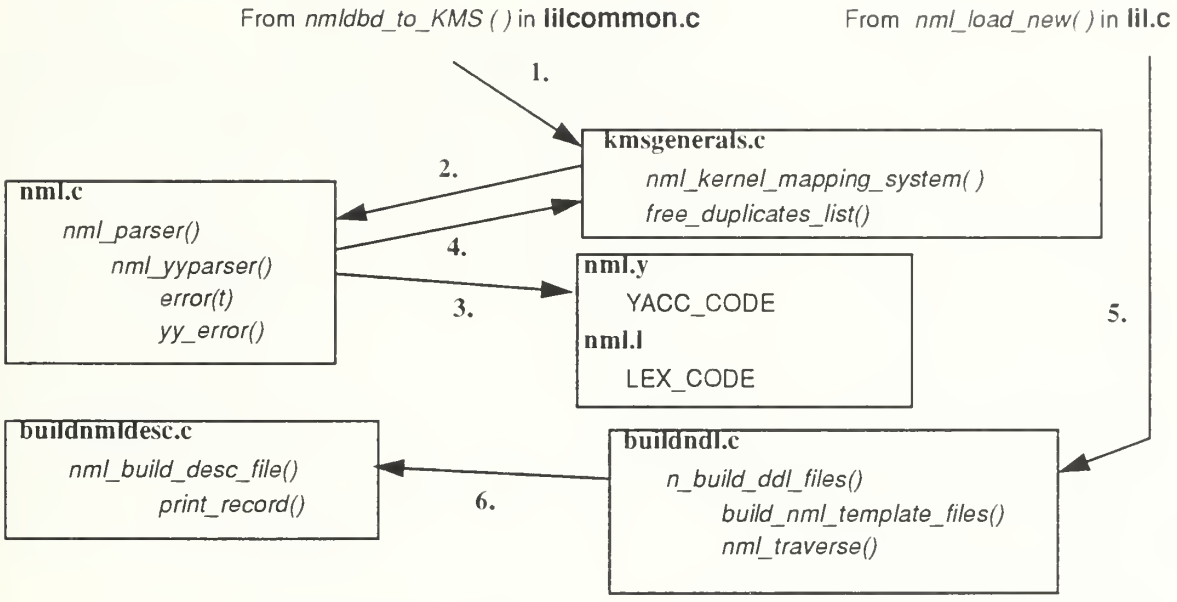
APPENDIX B. MODEL-LANGUAGE INTERFACE GENERIC FUNCTION MAPPING

Language Interface Layer



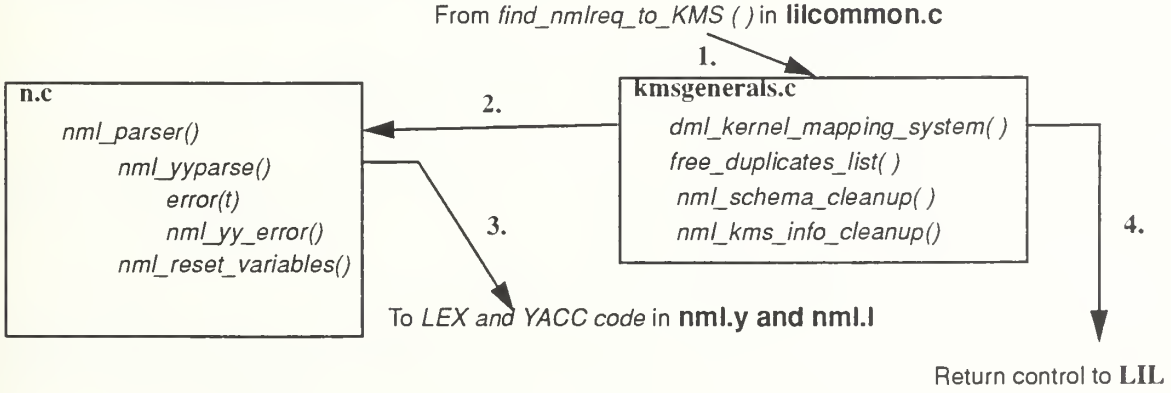
1. Establish new user by granting user id (for use with a mult-user system).
2. Call to **LIL** to allow user access to new model-language interface.
3. Whether processing an existing or new database, determine mode of input (terminal or file).
4. Call is made to either load a new database to **MDBS** or process an existing database
5. Call to **KMS** to parse the schema file.
6. Call to **KMS** to parse a request transaction selected by the user.
7. Call to **KC** to execute a transaction or load a schema parsed by **KMS** onto **MDBS**.

- *Loading a Schema File*



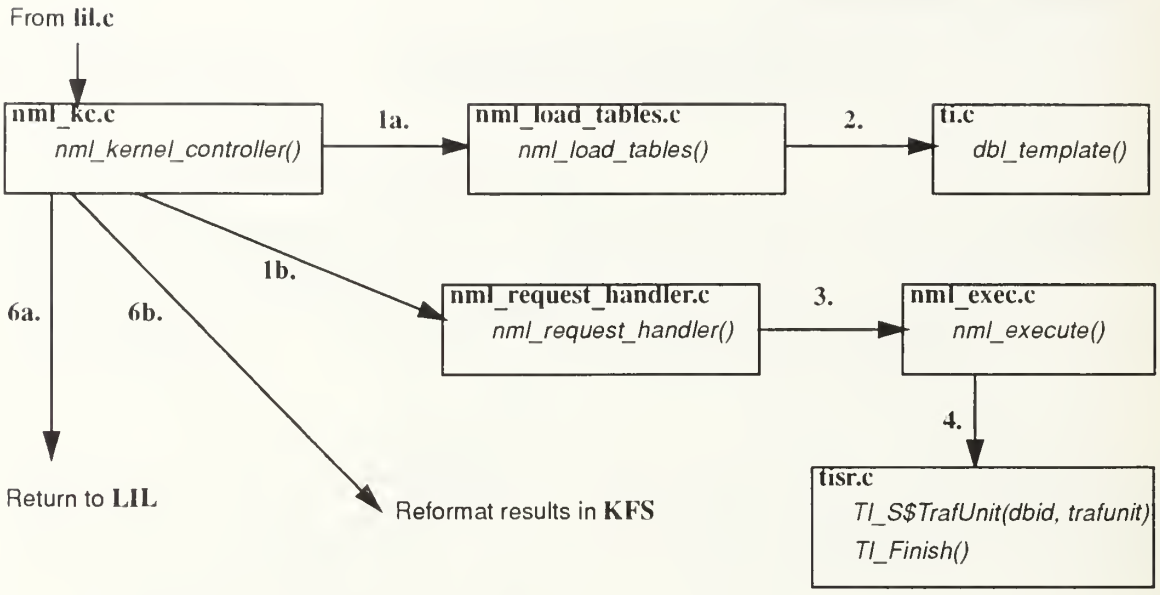
1. Function call from LIL to KMS to load the schema file.
2. KMS calls the parsing function.
3. Call LEX and YACC code to validate input stream
4. Control returned to kmsgenerals.c
5. Build the template file.
6. Build the descriptor file.

- *Loading a Request File*



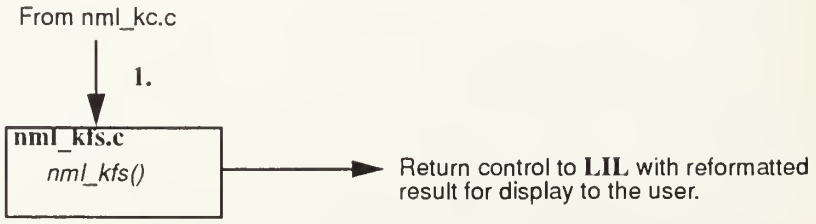
1. Function call from LIL to KMS to load the request file.
2. KMS call the parsing function.
3. Call LEX and YACC code to validate request transaction.
4. Transaction execution complete, return to LIL.

Kernel Controller



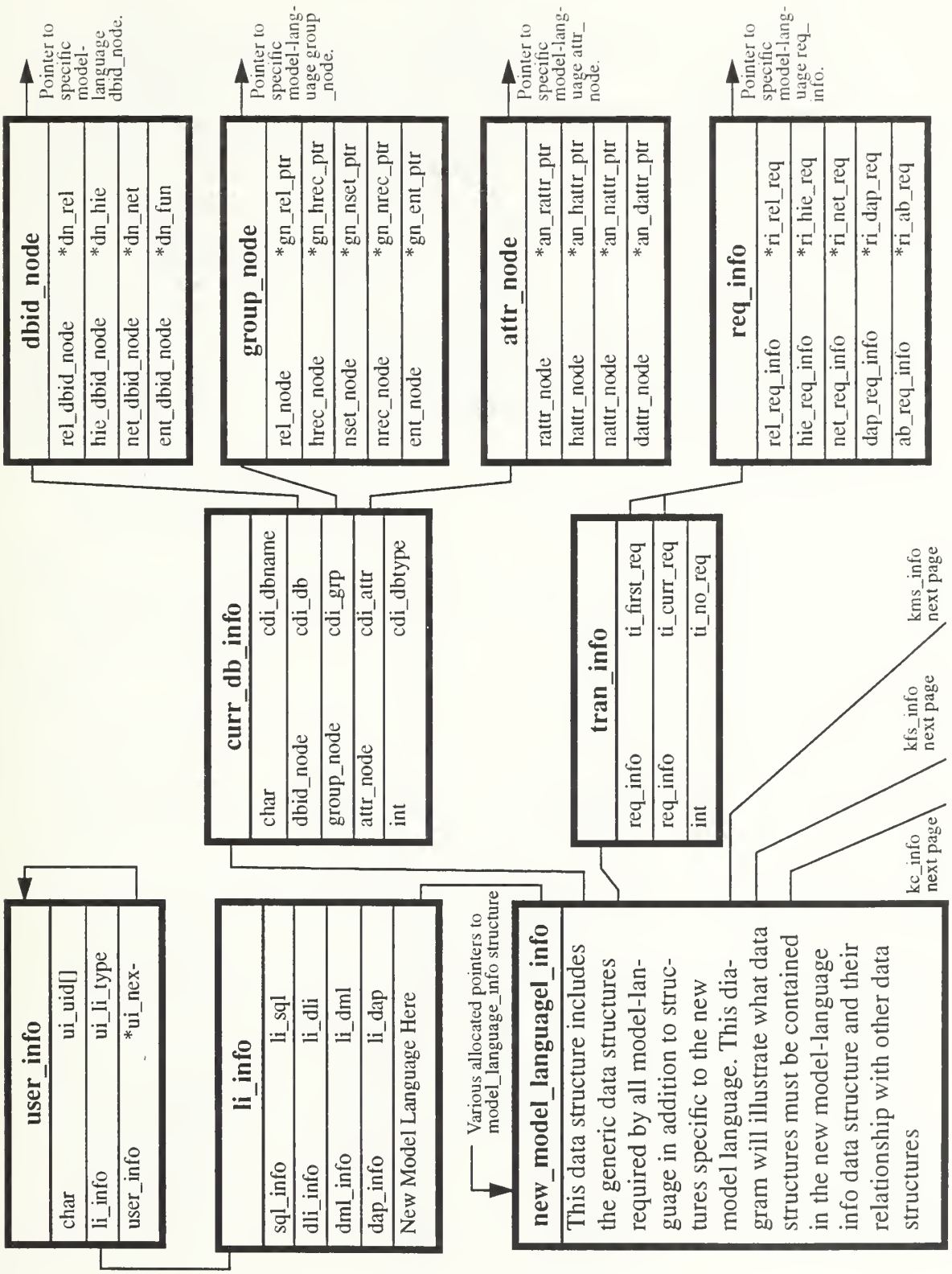
1a. If loading a new schema, calls the function to load template file.
1b. If loading a request, call appropriate request handler based on transaction type.
2. Call the Test Interface to load the template file.
3. Execute the transaction on **MDBS**.
4. Call the Test Interface to send transaction to **KDS**.
5a. Schema is loaded to **MDBS**, control returned to **LIL**.
5b. Transaction completes execution and must be reformatted by the **KFS**

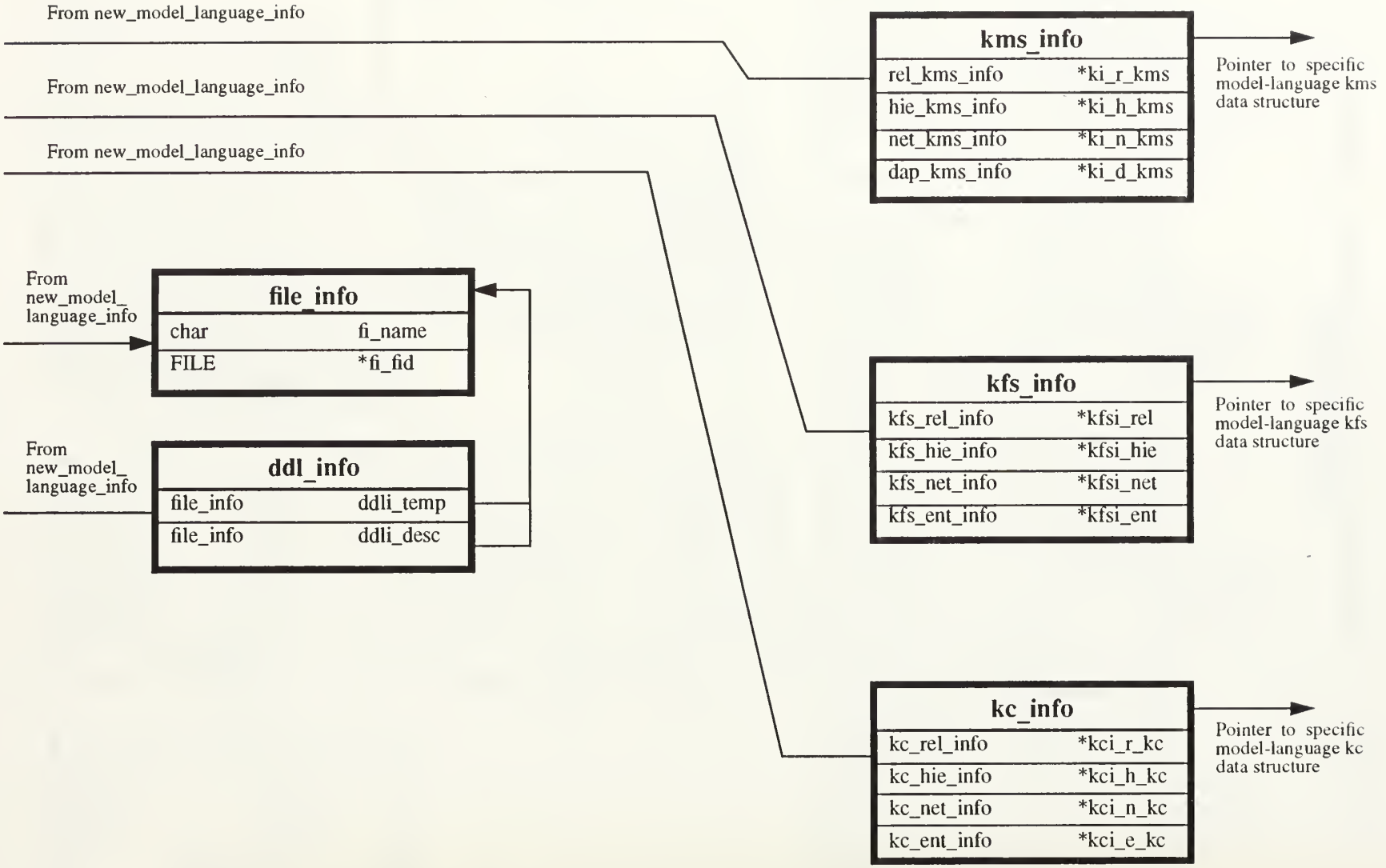
Kernel Formatting System



1. Receive control from **KC**; reformat **ABDL** result into **UDM** format

APPENDIX C. MM&MLDS GENERIC MODEL-LANGUAGE DATA STRUCTURE




From new_model_language_info
From new_model_language_info
From new_model_language_info
kms_info
rel_kms_info *ki_r_kms
hie_kms_info *ki_h_kms
net_kms_info *ki_n_kms
dap_kms_info *ki_d_kms
Pointer to specific model-language kms data structure
From new_model_ language_info
file_info
char fi_name
FILE *fi_fid
From new_model_ language_info
ddl_info
file_info ddli_temp
file_info ddli_desc
kfs_info
kfs_rel_info *kfsi_rel
kfs_hie_info *kfsi_hie
kfs_net_info *kfsi_net
kfs_ent_info *kfsi_ent
Pointer to specific model-language kfs data structure
kc_info
kc_rel_info *kci_r_kc
kc_hie_info *kci_h_kc
kc_net_info *kci_n_kc
kc_ent_info *kci_e_kc
Pointer to specific model-language kc data structure

APPENDIX D. GENERIC MAKEFILES FOR NEW MODEL-LANGUAGE INTERFACES

```
##############################################################################
# Title : Generic Makefile for a new model-language interface.               #
# file: Makefile                                                             #
# path:  db3 /usr/work/mdbs/NewVersion/CNTRL/TI/LangIF/src/NML/Makefile      #
##############################################################################

LIB= $(HOME)/lib/nml.a

HOME= /usr/work/mdbs/NewVersion/CNTRL/TI/LangIF
LPR= lpr
LPRFLAGS= -p
AR= ar
ARFLAGS= cr
RANLIB=ranlib

$(LIB): objects
        rm -f $(LIB)
        $(AR) $(ARFLAGS) $@ */*.o
        $(RANLIB) $@

quick:
        rm -f $(LIB)
        $(AR) $(ARFLAGS) $(LIB) */*.o
        $(RANLIB) $(LIB)

objects:
        cd Alloc; make
        cd Kc; make
        cd Kfs; make
        cd Kms; make
        cd Lil; make

clean:
        cd Alloc; make clean
        cd Kc; make clean
        cd Kfs; make clean
        cd Kms; make clean
        cd Lil; make clean

print:
        cd Alloc; make print
        cd Kc; make print
        cd Kfs; make print
        cd Kms; make print
        cd Lil; make print
```

*Replace **NewVersion** with the most current version name of MDBS software. This is where the new version of MDBS software will be stored.*

```
##########################################################################################
# Title : Generic Makefile for compiling all model-language interfaces.                  #
# file: makefile                                                                         #
# path: db3 /usr/work/mdbs/NewVersion/CNTRL/TI/LangIF/makefile                           #
##########################################################################################

SRC = /usr/work/mdbs/NewVersion/CNTRL/TI/LangIF/src/

quick:
#       cd $(SRC)Com; make
#       cd $(SRC)Dli; make
#       cd $(SRC)Dml; make
#       cd $(SRC)Sql; make
#       cd $(SRC)Dap; make
#       cd $(SRC)Obj; make
        cd $(SRC)NML; make

clean:
#       cd $(SRC)Com; make clean
#       cd $(SRC)Dli; make clean
#       cd $(SRC)Dml; make clean
#       cd $(SRC)Sql; make clean
#       cd $(SRC)Dap; make clean
#       cd $(SRC)Obj; make clean
#       cd $(SRC)NML; make clean;

print:
#       cd $(SRC)Com; make print
#       cd $(SRC)Dli; make print
#       cd $(SRC)Dml; make print
#       cd $(SRC)Sql; make print
#       cd $(SRC)Dap; make print
#       cd $(SRC)Obj; make print
#       cd $(SRC)NML; make print
```

To compile specific model-language interfaces, remove the # in front of the model-language to be compiled. To compile all model-language interfaces, remove the # from all the commands under **quick:** .

To delete all object files after compilation, remove the # in front of the model-language desired under the **clean:**.. *To print out the source code for a model-language, remove the # in front of the commands below* **print:** .

REFERENCES

[BENS 85] Benson, T. P., and Wentz, G. L., *The Design and Implementation of a Hierarchical Interface for the Multilingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.

[BROD 89] Brodie, J., and Plauger, P.J., *Standard C: Programmer's Quick Reference Series*, Microsoft Press, 1989.

[DEMU 87] Demurjian, S. J., *The Multi-Lingual Database System - A Paradigm and Test-Bed for the Investigation of Data-Model Transformations, Data-language Translations and Data-Model Semantics*, Ph.D. Dissertation, The Ohio State University, 1987.

[EMDI 85] Edmi, B., *The Implementation of a Network CODASYL - DML Interface for the Multilingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

[HALL 89] Hall, James E., *Performance Evaluations of a Parallel and Expandable Database Computer - The Multi-Backend Database Computer*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1989.

[HSIA 89] Hsaio, D. K., and Kamel, M. N., "Heterogeneous Databases: Proliferations, Issues, and Solutions", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 1, March 1989.

[HSIA 91] Hsiao, D. K., "A Parallel, Scalable, Microprocessor-Based Database Computer for Performance Gains and Capacity Growth", *IEEE Micro*, December 1991.

[HSIA 92] Hsiao, D. K., "Federated Databases and Systems: Part I - A Tutorial on Their Data Sharing", *VLDB Journal*, 1992.

[JOHN 78] Johnson, S. C., *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hills, New Jersey, July 1978.

[KELL 84] Kelly, A., and Pohl, I., *A Book on C - An Introduction to Programming in C*, The Benjamin/Cummings Publishing Company, Inc., 1984.

[KLOE 85] Kloepping, G. R., and Mack, J. F., *The Design and Implementation of a Relational Interface for the Multilingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.

[LESK 78] Lesk, H. E., and Schmidt, E., *Lex - A Lexical Analyzer Generator*, Bell Laboratories, Murray Hills, New Jersey, July 1978.

[SCNI 92] Shneiderman, B., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 2nd ed., pp. 11, Addison-Wesley Publishing Company, Inc., 1992.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
Dudley Knox Library Code 052 Naval Postgraduate School Monterey, California 93943	2
Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, California 93943	2
Director Training and Education MCCDC Code C46 1019 Elliot Road Quantico, Virginia 22134-5027	1
Professor David K. Hsiao Computer Science Department Naval Postgraduate School Monterey, California 93943	2
Captain Paul A. Bourgeois, USMC 103 Adventure Trail Cary, North Carolina 27513	1

144-403

DEMCO

AMERICAN BIND
9-93
10

DUDLEY KNOX LIBRARY
3 2768 00035890 7